

Course 397

C++ Best Practices and Design Patterns

by

Ulf Bilting

Technical Editor: Jasper Kent

LEARNING TREE INTERNATIONAL

Copyright

© LEARNING TREE INTERNATIONAL, INC.
All rights reserved.

All trademarked product and company names are the property of their respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or translated into any language, without the prior written permission of the publisher.

Copying software used in this course is prohibited without the express permission of Learning Tree International, Inc. Making unauthorized copies of such software violates federal copyright law, which includes both civil and criminal penalties.

Acknowledgements

The author would like to acknowledge the following for their contributions to this course

- John Ballam
- Jim Burgess
- Greg Davidson
- Richard Howells
- Magnus Karlsson
- Jasper Kent
- Learning Tree Product Management team



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



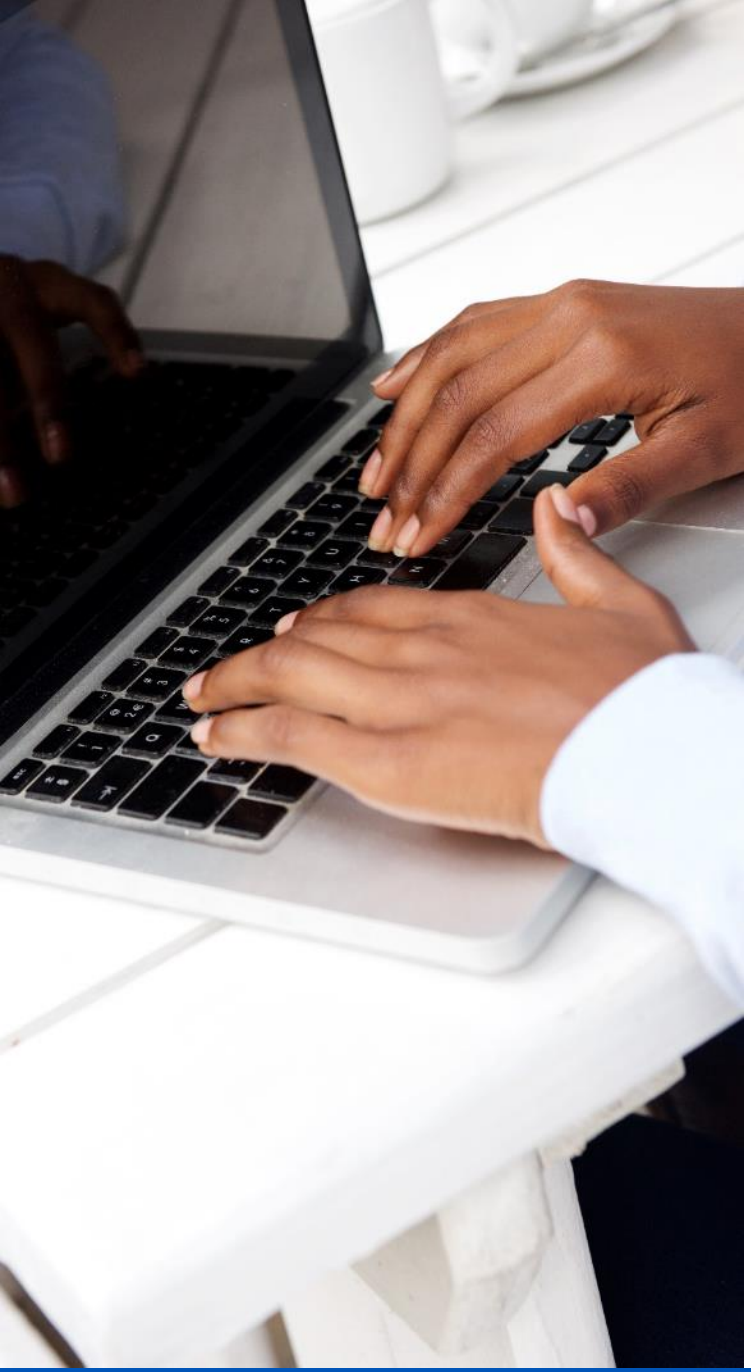
Introduction and Overview

LEARNING TREE INTERNATIONAL

Course Objectives

- **Use all significant features of the C++ language**
  - Effective use of classes, inheritance, exceptions, casting, templates, memory management, `virtual` destructors, `explicit`, `mutable`, and `const`
  - New in C++11/14/17/20: Lambdas, auto declarations, new initialization syntax, r-value references and move semantics, controlling default constructors, assignment with `=default`/`=delete`, concepts, and optional values
- **Apply design patterns to produce high-quality object-oriented C++ code**
  - Well-organized reuse of expert knowledge and experience
  - Singleton, Adapter, Template Method, Observer, Composite, Flyweight, Visitor, Factory, etc.
- **Make effective use of libraries**
  - Standard Library: standard templates, containers, iterators, algorithms, strings, streams, threads, smart pointers, memory management, etc.
  - Boost open-source libraries: serialization

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercises

Exercise Manual

- **The exercises are designed to reinforce the topics covered in the notes**
  - Chance to talk through ideas with your neighbor
  - Opportunities to learn from your neighbor
- **Many of the exercises are based on a project that simulates a pedestrian light-controlled crossing**
  - Designed to allow many aspects of C++ to be explored in a context you become familiar with as the course progresses
  - Some exercises use alternative projects

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Prerequisites

- **Basic C++ programming skills**
- **Some knowledge of object-oriented concepts**
- **Except for specific areas, this course does not attempt to teach basic C++ syntax or semantics**
  - Please see your instructor if you think you may be over- or under-qualified
  - However, don't hesitate to ask about areas in C++ that you are unsure about; these are often shared by others



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Contents

	Introduction and Overview
Chapter 1	C++ and Object-Oriented Development
Chapter 2	Patterns, Part 1: Design Patterns
Chapter 3	The Standard Library, Part 1: The STL
Chapter 4	Patterns, Part 2: Observer Pattern
Chapter 5	The Standard Library, Part 2: Strings, Streams, and Threads
Chapter 6	Storage Management
Chapter 7	Patterns, Part 3: Composite and Visitor Patterns
Chapter 8	Course Summary
	Next Steps

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Contents

Appendix A	**Description of Pelican Crossing and Running the Pelican Program on Unix/Linux**
Appendix B	**Overloading `new` and `delete`** **Memory Layout and Inheritance**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Use of the Standard Library

- **Throughout the course, we use the C++ Standard Library**
- **Facilities of the Standard Library are accessed differently than similar facilities in proprietary libraries**
  - This allowed programs to gradually migrate to the Standard Library
  - Here is a “Hello, World” program that uses the Standard Library

```
#include <iostream>
using namespace std;

int main(){
  cout << "Hello, World";
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

C++ 2011/2014/2017/2020 Standards

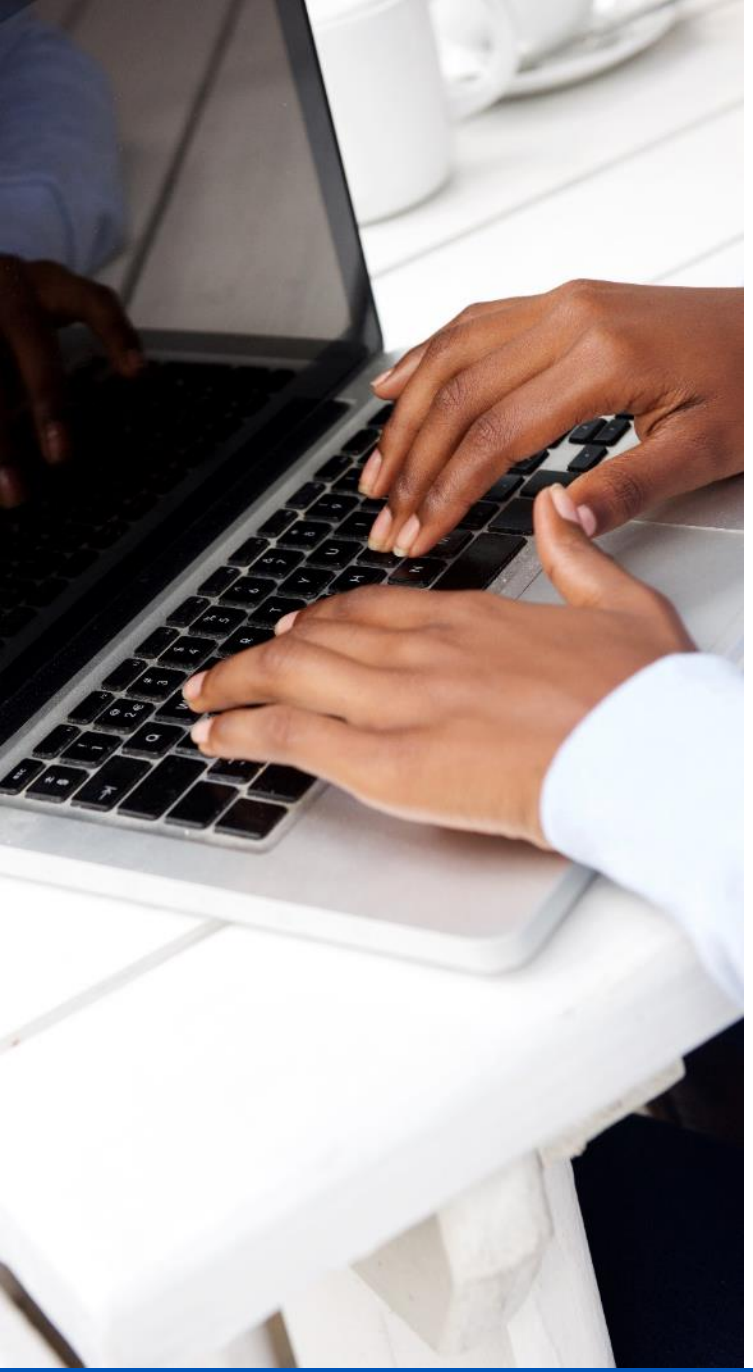
- **C++20 is the current standard, published in 2020**
  - It is very backward-compatible
- **It is not yet firmly established**
  - Many compilers still do not implement all new features
  - Many conservative programmers exist
- **C++11 was the big game changer**
  - 14/17/20 added nice, but less important features (author's opinion)
- **This course covers many of the new C++ features**
  - They are introduced in appropriate sections and marked as C++11/14/17/20
- **Old-style, traditional C++ code deliberately remains in some exercises. Update the code as you spot it!**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Exercises

- **There are suggested solutions available**
  - As separate runnable projects
  - Studying/comparing them might be a good idea
  - You can take home all exercises and solutions
- **Additional exercises**
  - Some additional exercises are described
  - You will be equipped to attempt these at any time after finishing the main exercise
  - You may not have time to finish these exercises—find some extra time at breaks, before or after class
- **Using Visual Studio**
  - All you need to know about using this C++ development environment is in this course
  - We focus on the C++ language, not a particular tool

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



The Exercises: Best Practice

Exercise Manual

- **You are all different**
  - Do the exercises in a way that is optimal for yourself
    - Don’t follow instructions blindly—reflect and discuss!
  - Ask for help any time
    - Partner or neighbor
    - Instructor
    - Look at solutions
  - Don’t get stuck for too long on any issue (three-minute rule)
    - Particularly not on anything uninteresting!
  - Experiment freely
    - You are not going to break anything (if you do, we’ll fix it)

Exercise Manual

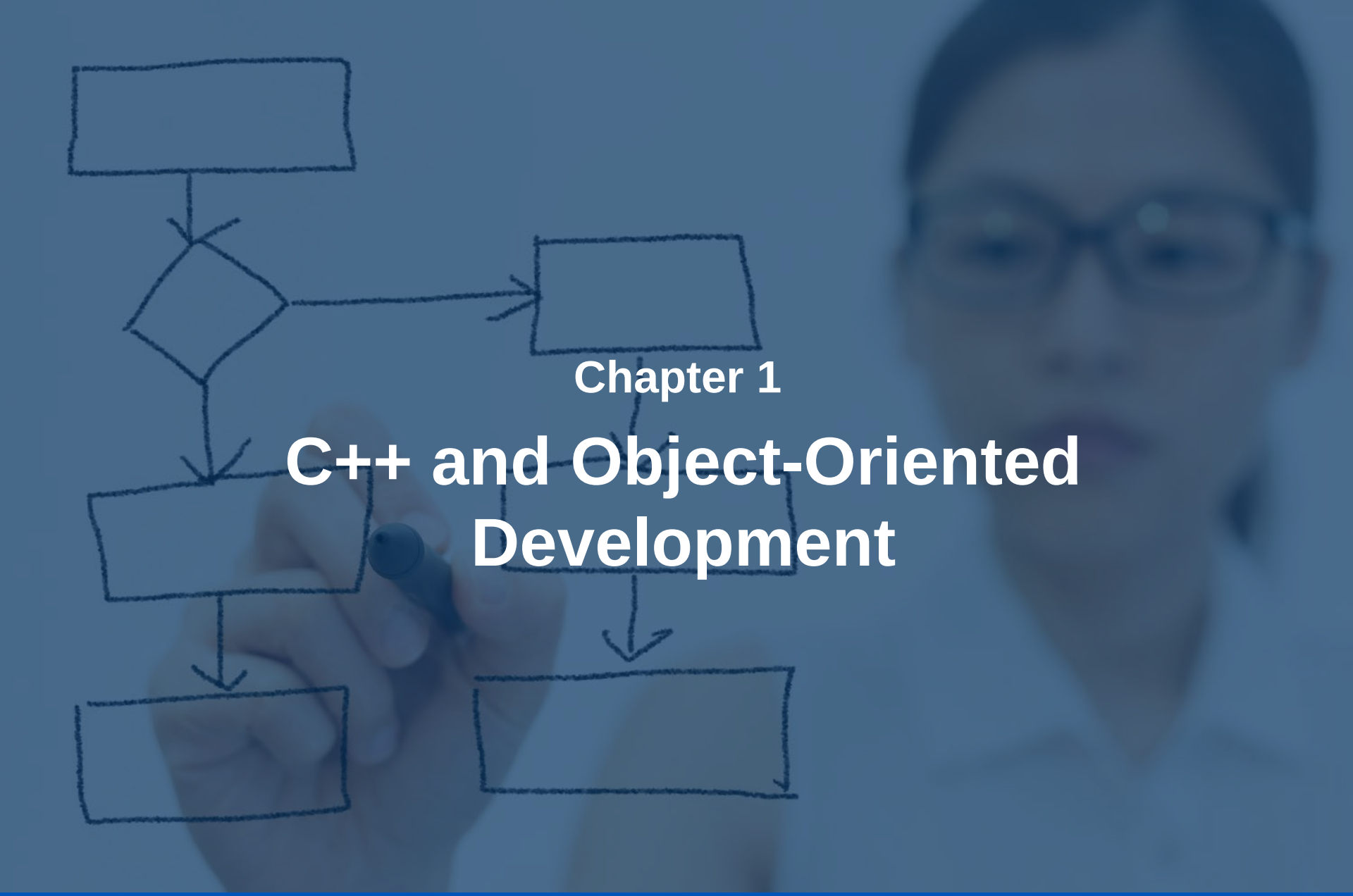
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Your Computer

- **Your course PC runs Microsoft Windows and is loaded with the following**
  - Microsoft Visual Studio
  - Boost C++ libraries
  - Documentation of the C++ standard and Boost libraries
- **The course is not about a particular environment, operating system, or development system**
  - All focus is on C++ itself and nonproprietary libraries



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



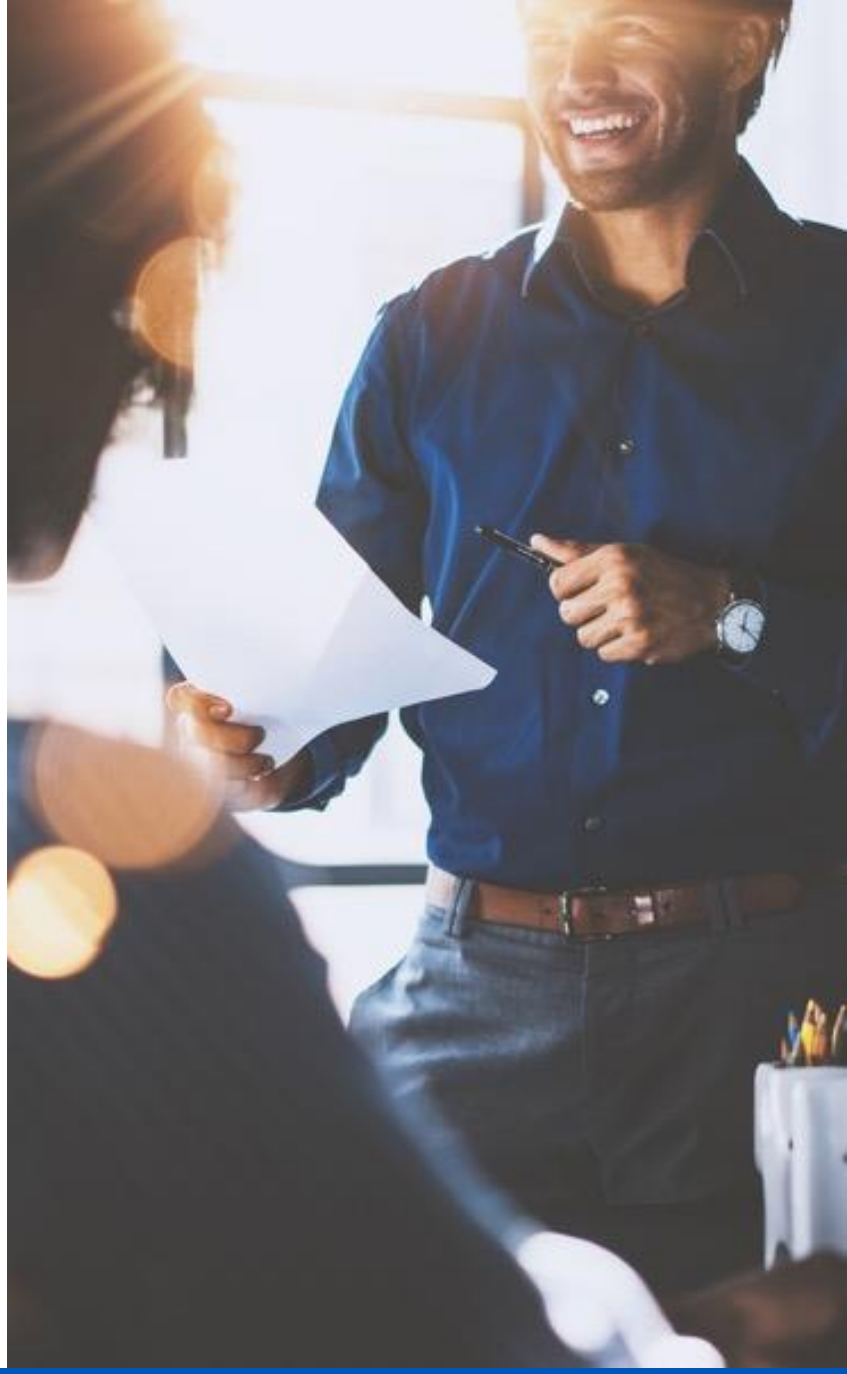
Chapter 1

C++ and Object-Oriented Development

LEARNING TREE INTERNATIONAL

Objectives

- **Learn the concepts underlying OO design and programming**
  - Using UML to annotate OO designs
  - How C++ implements these concepts
- **Learn how to take advantage of modern tools for C++ development**
  - Visual Studio



OO = object-oriented
UML = Unified Modeling Language

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

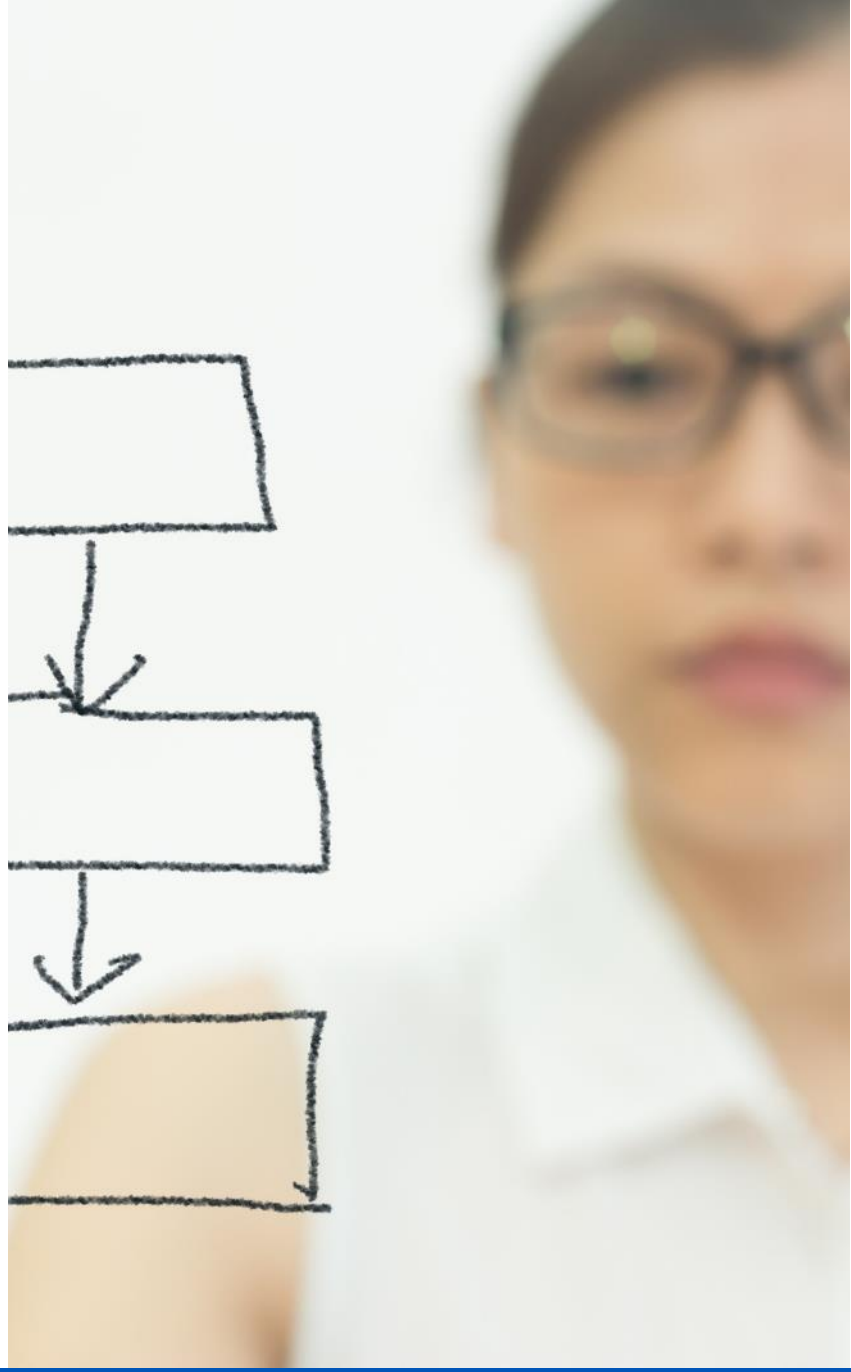
Contents

The OO Paradigm, UML, and C++

Classes, Attributes, and Methods

- Objects, Messages, and Associations
- Aggregation
- Implementation Inheritance
- Interface Inheritance and Polymorphism

- **Development Tools**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Classes, Attributes, and Methods

- **The *class* is the primary construct in OO**
- **A class describes a user-defined type**
  - *Attributes* hold the information required by each object of the class
  - *Methods* embody the behavior of an object of the class
- **How does C++ implement attributes and methods?**
  - For example, a lightbulb
    - What information should a lightbulb object hold?
    - What might its methods be?
  - OO wisdom states that attributes should be hidden within an object
    - How does C++ support this?
    - Why is this important?

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Classes, Attributes, and Methods in C++: `Lightbulb` Class

```
// lightbulb.h
class Lightbulb {
public:
  enum class State {ON, OFF, BROKEN};
  enum class Color {WHITE, RED, GREEN, BLUE};

  void setState(State s);
  State getState() const;
  bool isBroken() const;

  Lightbulb(Color c=Color::WHITE);
private:
  State myState;
  const Color myColor;
};
```

C++11: New scoped enum



```
// lightbulb.cpp
#include "lightbulb.h"

void Lightbulb::setState(State s)
{
  myState = s;
}

...
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aside: enum and enum class

```
enum State {ON, OFF, BROKEN};                    // old
enum class Color {WHITE, RED, GREEN, BLUE};      // C++11
// both get values 0,1,2...

int s = ON;  // not scoped and implicit conversion to int
int c = static_cast<int>(Color::WHITE);
             // scoped and needs explicit conversion to int

bool b = ON < 45;            // Oops, not intended but correct
bool d = Color::WHITE < 45; // Compile error

State st = Color::WHITE;     // Compile error
Color co = ON;               // Compile error
                             // Both are type safe at assignment

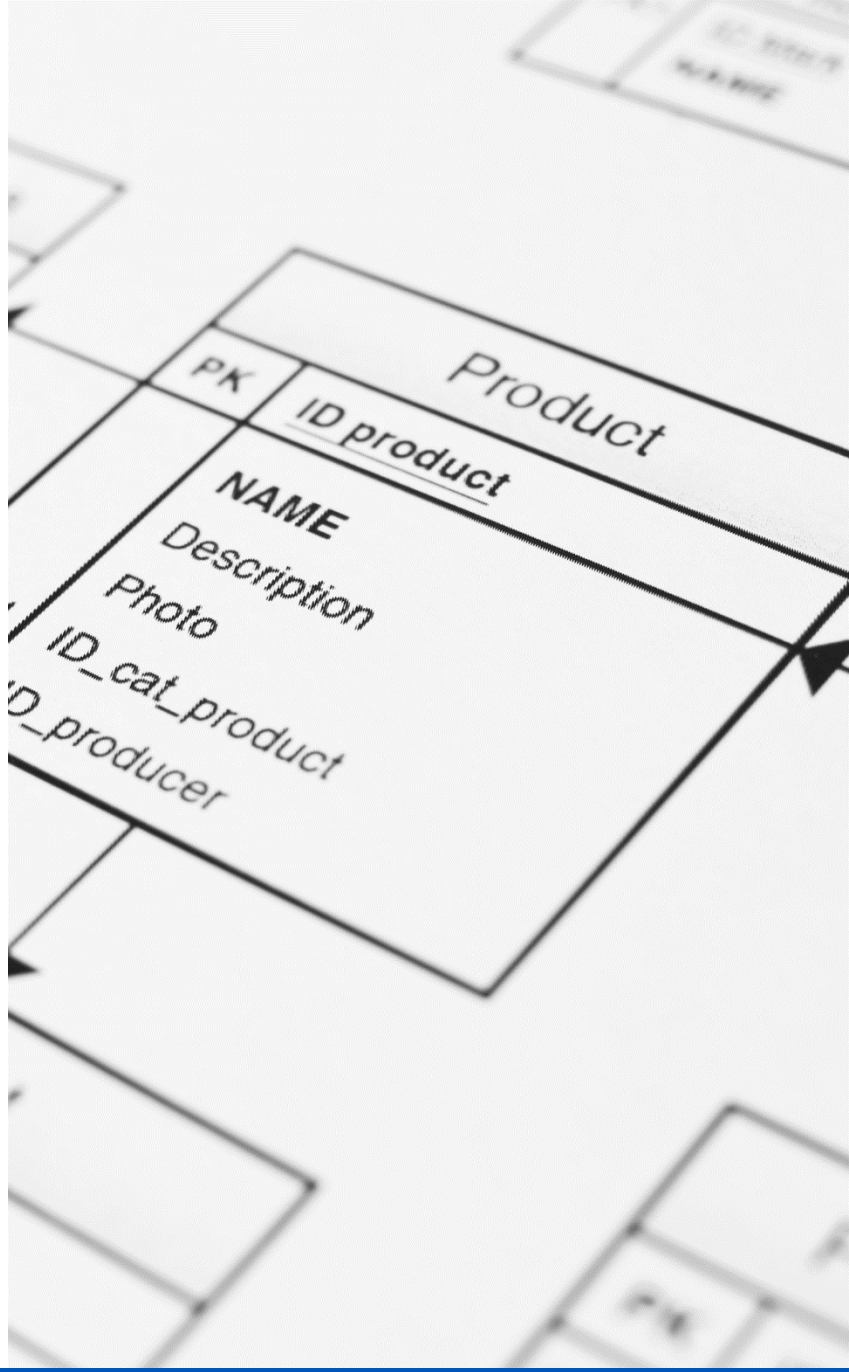
enum class Color2 : unsigned char {PURPLE, BLACK, GREY};
                             // new enum may specify storage type

// New enum class generally safer, but if you really want
// to use enum values as int, the old one is more practical.
// In exercises, we will use the old one for this reason
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing UML

- **UML is the de facto standard graphical notation for describing OO systems**
  - Developed by Grady Booch, James Rumbaugh, and Ivar Jacobson



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing UML

- **UML specifies a set of diagrams covering:**

Diagram	Purpose
Use-case diagram	Requirement capture
Class diagram Object diagram	Class modeling
Sequence diagram Collaboration diagram State diagram Activity diagram	Behavioral modeling
Component diagram	Logical partitioning
Deployment diagram	Physical partitioning

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

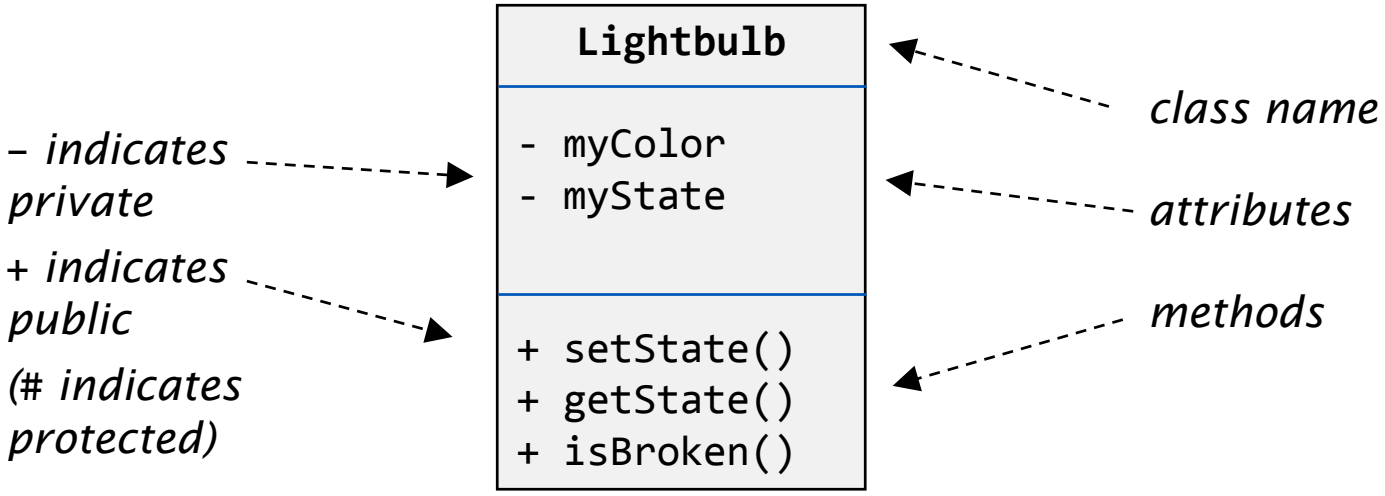
Introducing UML

- **UML specifies a graphical notation, but does not specify a *process***
  - A process is a set of steps to be followed
  - You have the freedom to tailor the process to a particular application
- **This course shows how the notation maps to the constructs of C++**
  - Provides an intuitive, graphical tool for comprehending the structure of C++ programs
- **UML is a flexible tool**
  - Details may be left out
- **Opinions vary among experts**
  - "Brilliant tool for communicating OO designs; most people prefer graphical descriptions" vs. "Ad-hoc graffiti with fuzzy meaning"
  - Be aware how easy it can be to misinterpret pictures!
- **Should every little class be drawn in UML?**
- **Should all UML continuously be updated in parallel with the code?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Classes, Attributes, and Methods in UML

- **Here is the UML notation for our `Lightbulb` class**
  - Can show parameters, return types, default arguments, etc.
  - These need not appear on diagrams



- **Could this diagram possibly be interpreted as denoting a single object, rather than a class?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

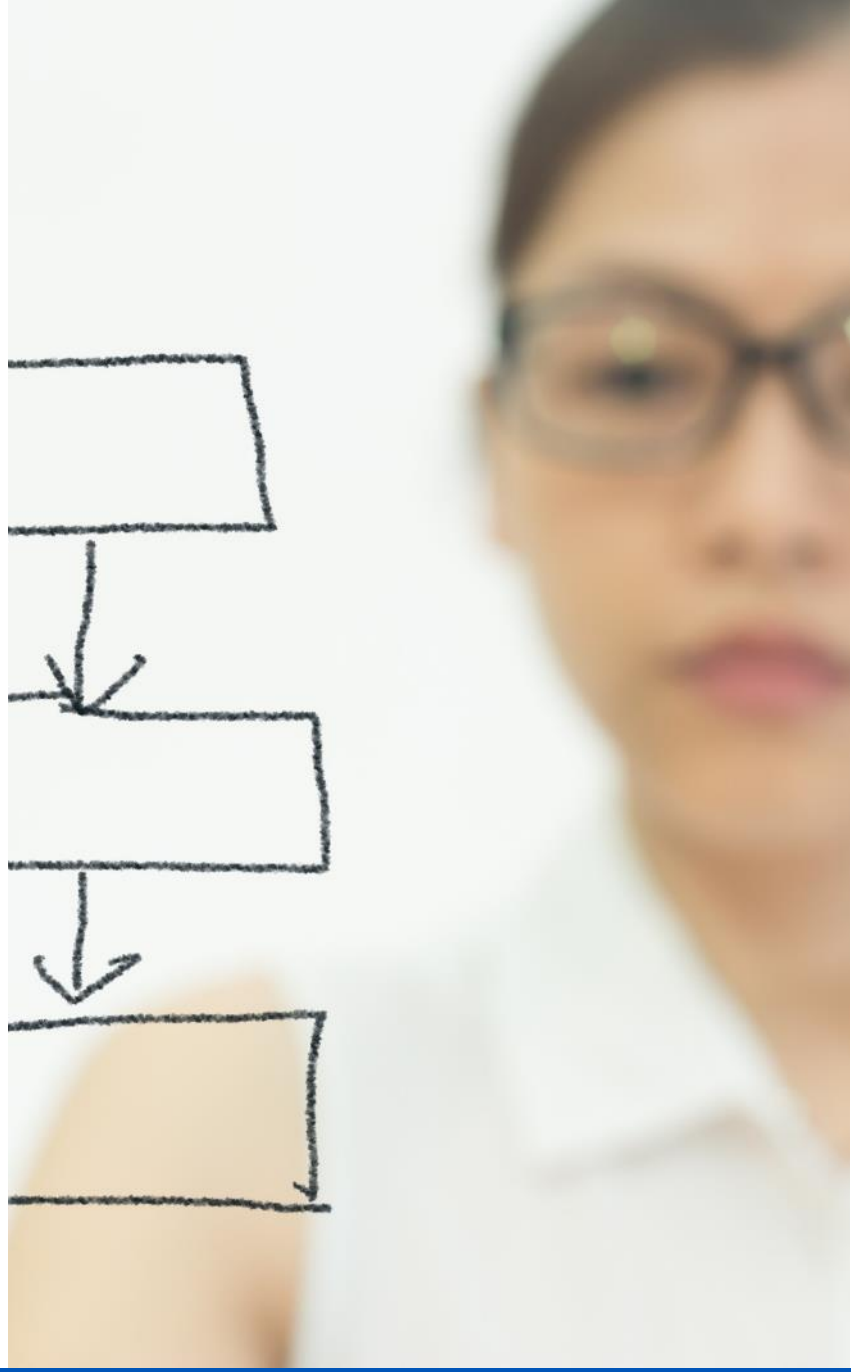
The OO Paradigm, UML, and C++

- Classes, Attributes, and Methods

Objects, Messages, and Associations

- Aggregation
- Implementation Inheritance
- Interface Inheritance and Polymorphism

- **Development Tools**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objects, Messages, and Associations

- **An object is an instance of a class**
- **Methods are invoked by one object sending a *message* to another object**
- **An object can remember *links* to other objects**
  - The socket in my kitchen holds a particular lightbulb
- **A link is an instance of an *association* between classes**
  - Sockets hold lightbulbs
- **How do we implement messages, associations, and links in C++?**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Associations in C++: Socket Class

```
// socket.h
class Lightbulb;
class Socket {
public:
  Lightbulb* changeBulb(Lightbulb* b);
  void applyPower(bool);
  bool isPowerOn() const;
  bool isLightBroken() const;
private:
  bool powerOn;
  Lightbulb* myLight;
};
```



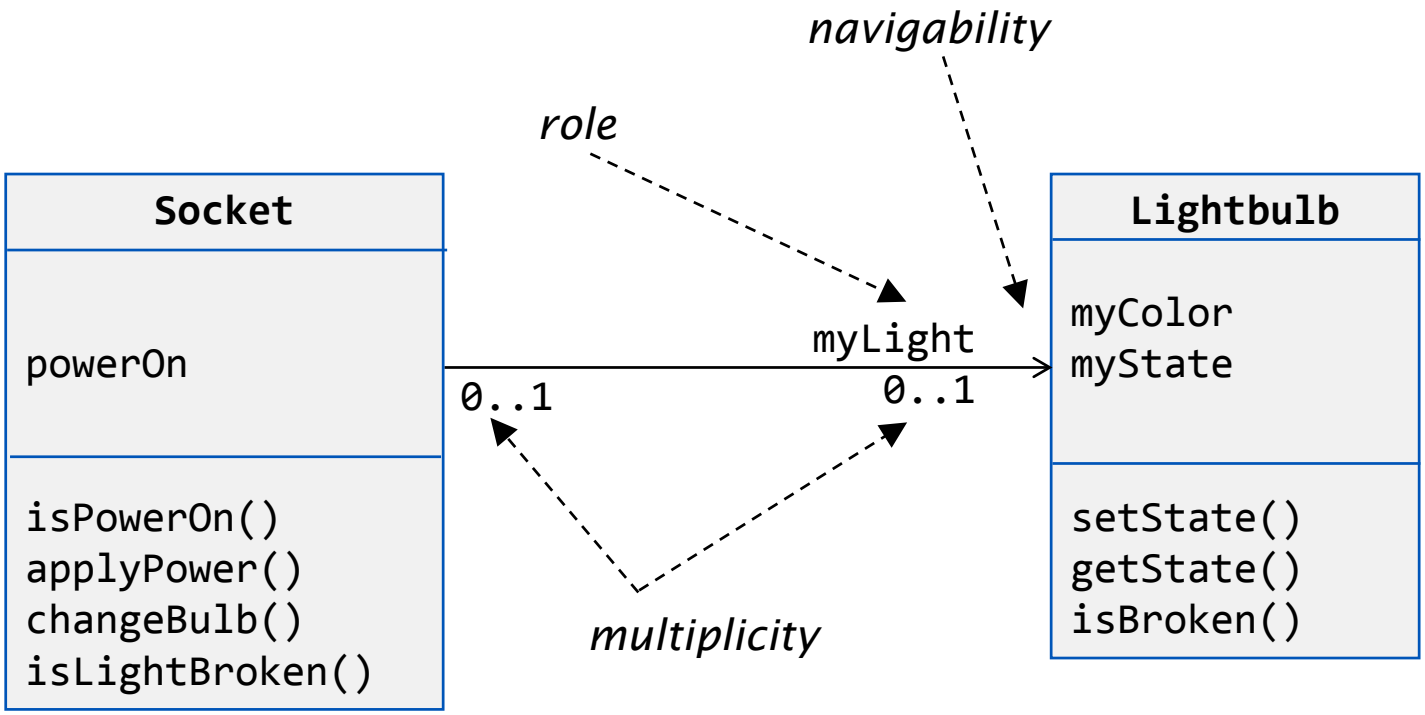
```
// socket.cpp
#include "socket.h"
Lightbulb* Socket::changeBulb
                      (Lightbulb* b){
  Lightbulb* old = myLight;
  myLight = b;
  return old;
}

...
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Associations in UML

- **All elements except the line are optional**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objects and Messages in C++: Changing a Lightbulb

```
#include "socket.h"
#include "lightbulb.h"
int main(){
  Socket s;
  s.changeBulb(new Lightbulb);
  ...
  if(s.isLightBroken())
  {
    Lightbulb* oldBulb;
    oldBulb = s.changeBulb(new Lightbulb);
    delete oldBulb;
  }
}
```





```
// socket.cpp
...
bool Socket::isLightBroken() const{
  return myLight->isBroken();
}
```

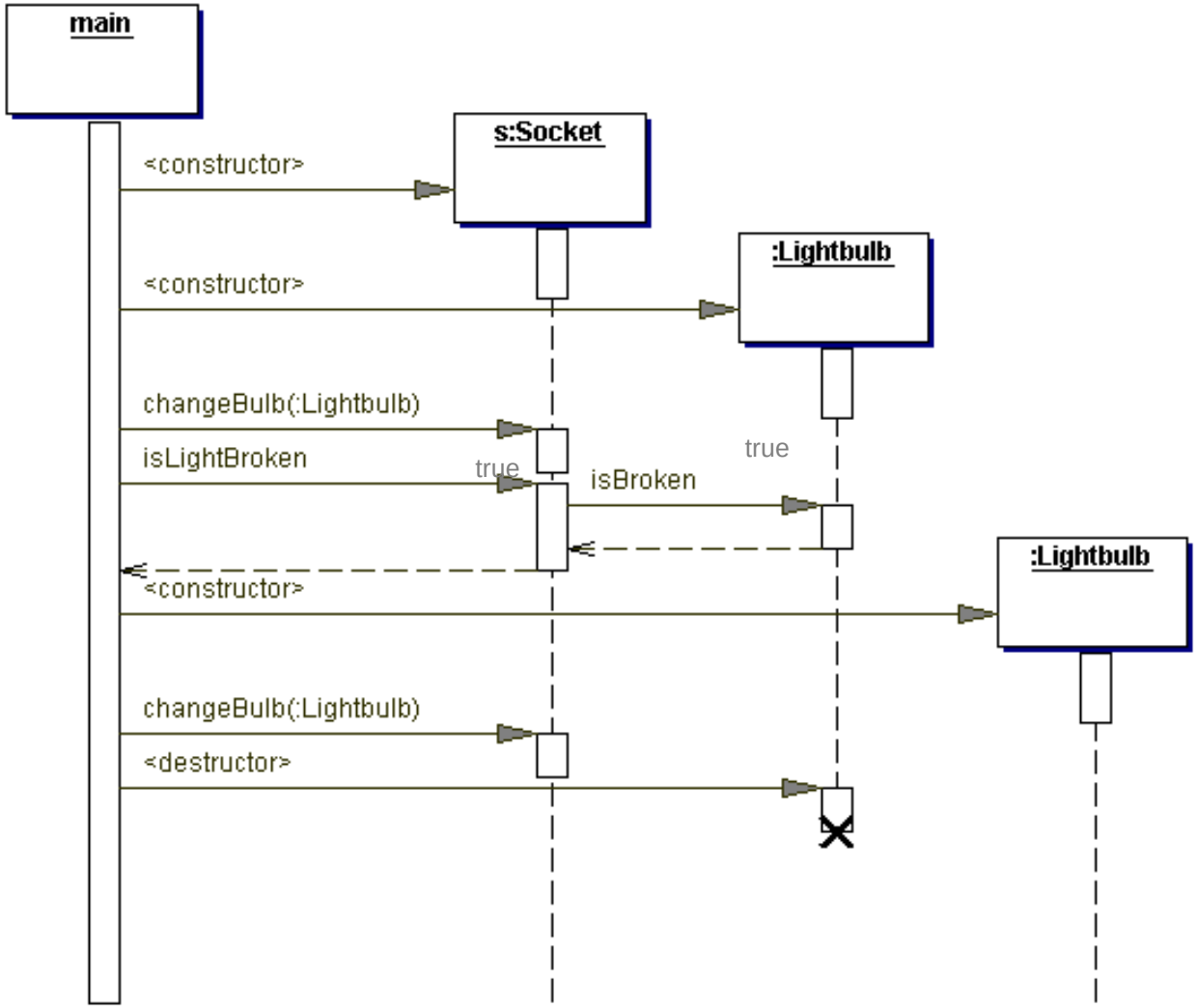
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objects and Messages in UML: The Sequence Diagram

- **A sequence diagram shows a time-ordered sequence of events**
  - Time flows down the page
  - Return from function shown as separate event
- **Useful for showing complex interaction between several objects**
- **Full sequence diagram notation can show**
  - Conditional branches
  - Concurrent activity in one or more objects
  - Events taking significant time to transmit
  - Timing constraints for methods and messages

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objects and Messages in UML: The Sequence Diagram



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

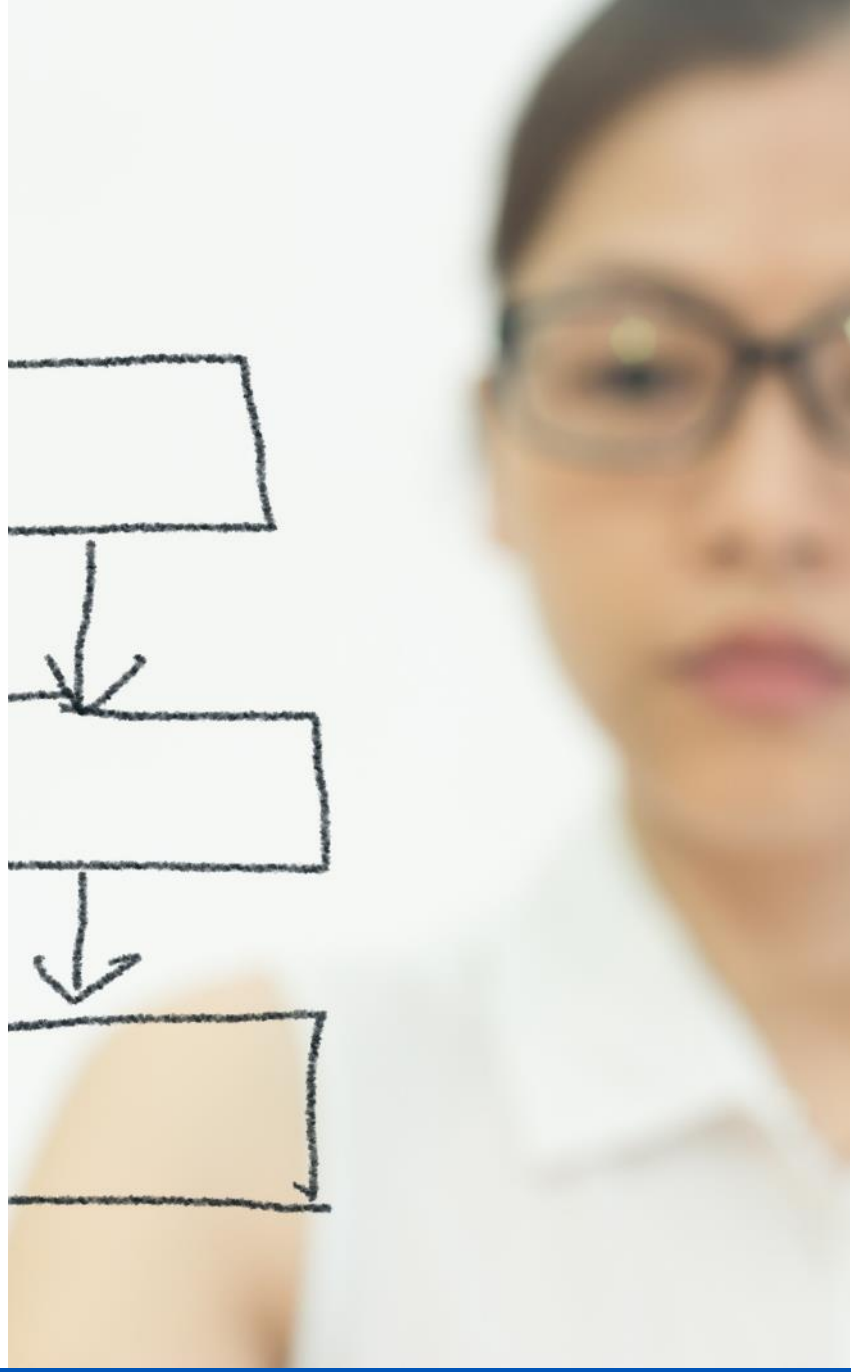
The OO Paradigm, UML, and C++

- Classes, Attributes, and Methods
- Objects, Messages, and Associations

Aggregation

- Implementation Inheritance
- Interface Inheritance and Polymorphism

- **Development Tools**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aggregation

- **Very often, an object is made up of smaller parts**
- **This is called *aggregation* or *composition***
  - Client may treat the *aggregate* as a single object
  - Composite may *delegate* method calls to its *components*
- **For example, a traffic light is an aggregation of light sockets**
  - A controller object tells the traffic light to switch to green
  - The traffic light controls the power to each of its sockets
- **The socket is a part of the traffic light, whereas the bulb is not part of the socket**
  - This is what distinguishes aggregation from association



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aggregation in C++

- **Aggregation implemented using embedded objects**
  - When components are born and die with the main object, the aggregation form is called *composition*

```
#include "socket.h"

class TrafficLight {
public:
  enum class State {GREEN, YELLOW, RED};
  void setState(State s);
  ...
private:
  Socket redSocket;
  Socket yellowSocket;
  Socket greenSocket;
};
```

- **Aggregation can also be implemented with pointers or references**
- **Why would we implement aggregation/composition with pointers or references?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

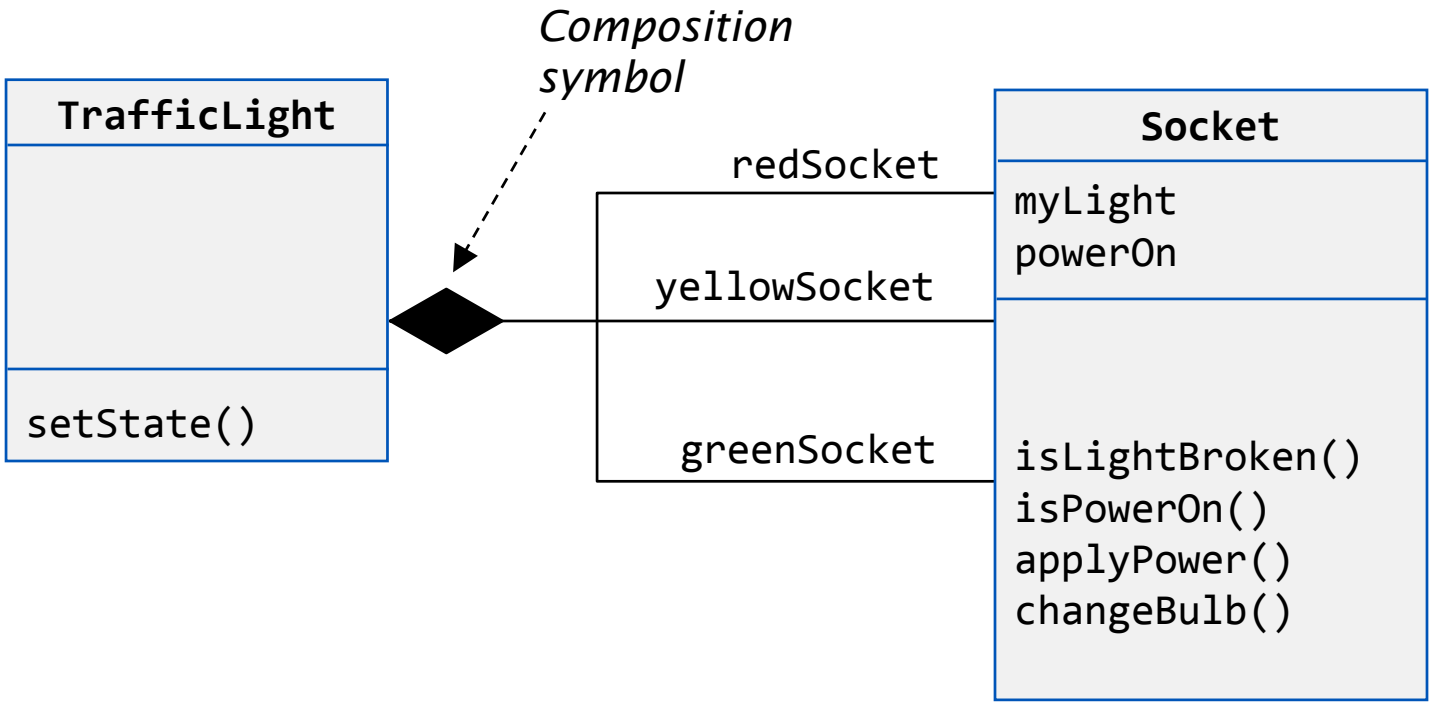
Aggregation in C++

- **Aggregate methods can delegate to components**
  - Not always merely a mechanical process
  - Aggregate method may embody coordinating logic

```
// trafficLight.cpp
...
void TrafficLight::setState(State s){
  switch(s){
  case State::RED:
    redSocket.applyPower(true);
    yellowSocket.applyPower(false);
    greenSocket.applyPower(false);
    break;
  case State::YELLOW:
    ...
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Composition/Aggregation in UML



- **Composition is the stronger form of aggregation**
  - The parts are born and die with the owning object
  - Normal aggregation is denoted by an unfilled rhombic symbol
  - One can easily engage in hair-splitting arguments trying to decide between aggregation and composition

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

AdaptaLearn™ Enabled

- **Electronic, interactive exercise manual**
- **Offers an enhanced learning experience**
  - Some courses provide folded steps that adapt to your skill level
  - Code is easily copied from the manual
  - After class, the manual can be accessed remotely for continued reference and practice



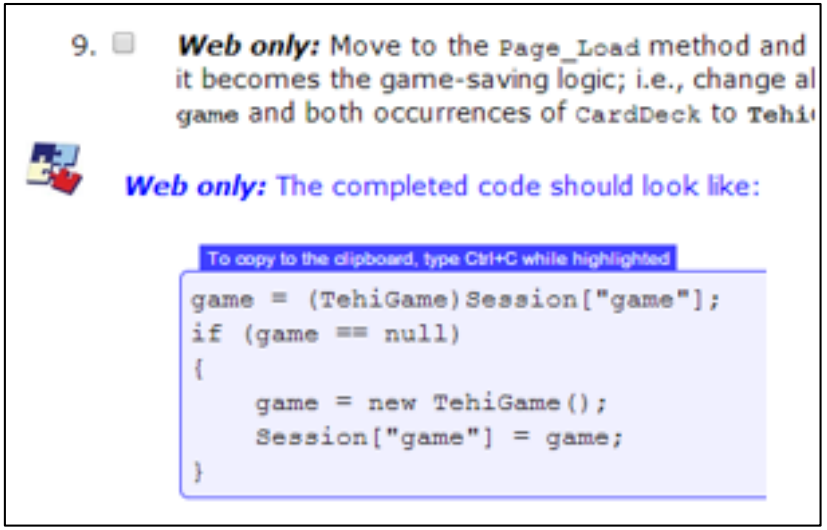
- **Printed and downloaded copies show all detail levels (hints and answers are unfolded)**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

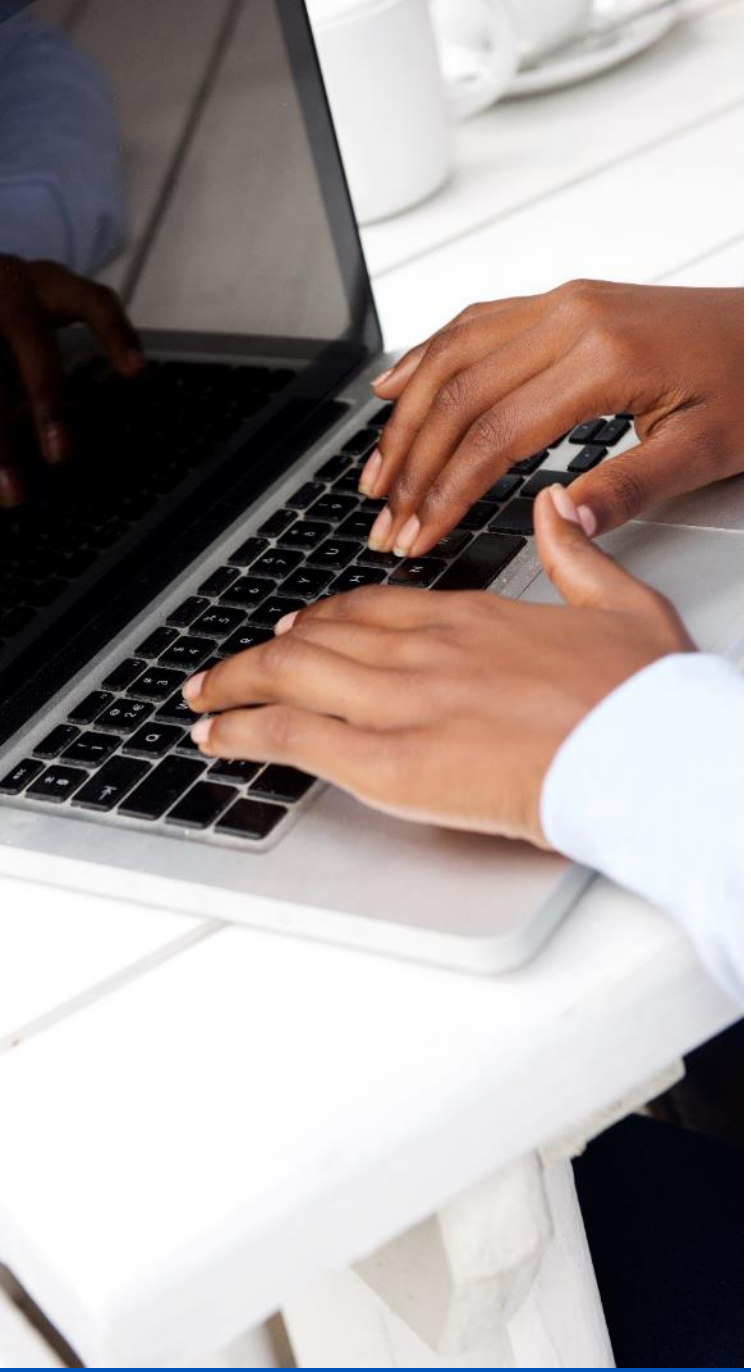
Using AdaptaLearn™

Do Now

1. **Launch AdaptaLearn by double-clicking its icon on the desktop**
   - Move the AdaptaLearn window to the side of your screen or shrink it to leave room for a work area for your development tools
2. **Select an exercise from the exercise menu**
   - Zoom in and out of the AdaptaLearn window
   - Toggle between the AdaptaLearn window and your other windows
3. **Look for a folded area introduced with blue text (not available in all courses)**
   - Click the text to see how folds work
4. **Try to copy and paste text from the manual**
   - Some courses have code boxes that make it easy to copy areas of text while highlighted (as shown)



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 1.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 1.1: Using the Toolset

In this exercise, you will be introduced to

- Visual Studio, an integrated development environment
- The course theme project: the pedestrian light-controlled crossing
- Finding and fixing a bug with the help of C++11-scoped `enum`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

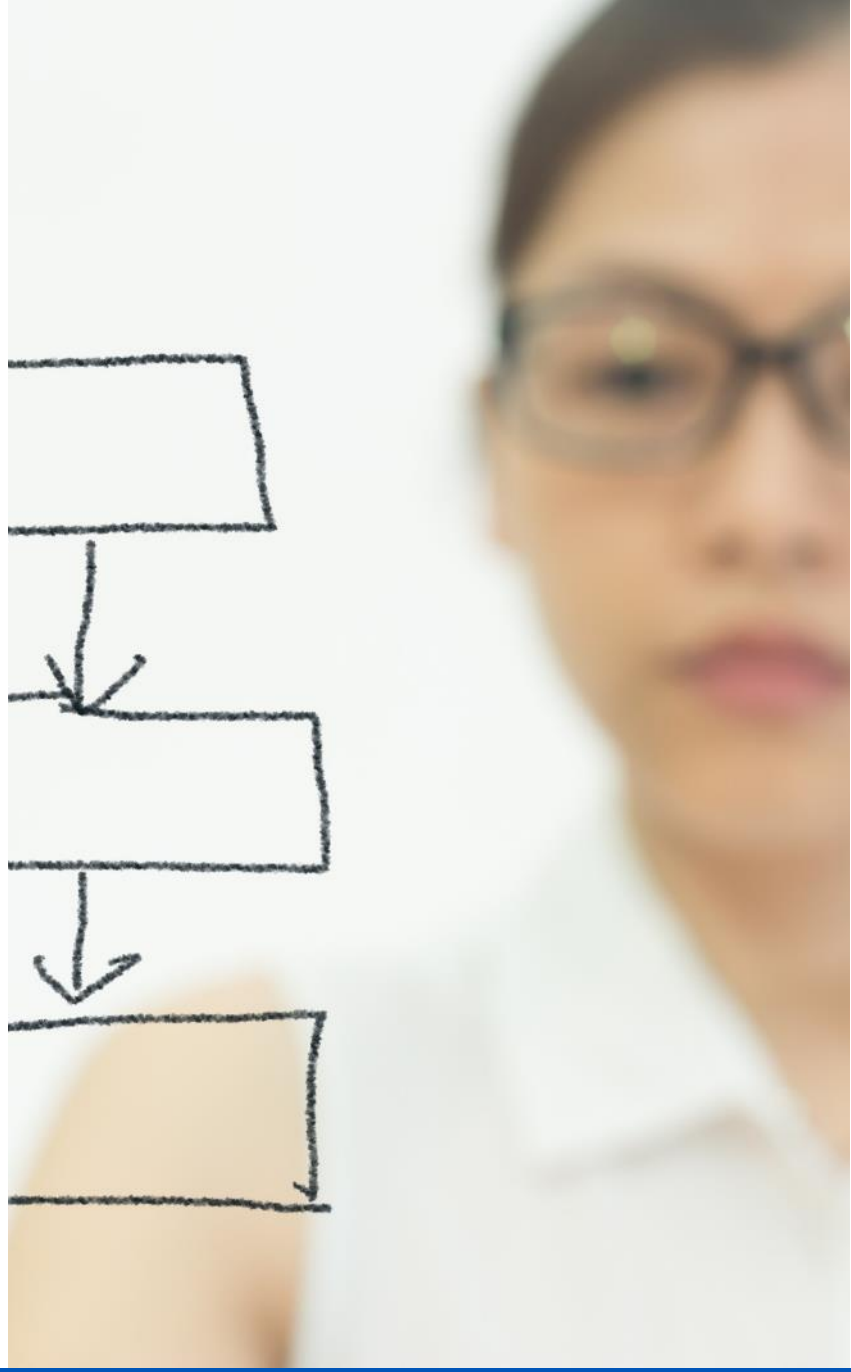
The OO Paradigm, UML, and C++

- Classes, Attributes, and Methods
- Objects, Messages, and Associations
- Aggregation

Implementation Inheritance

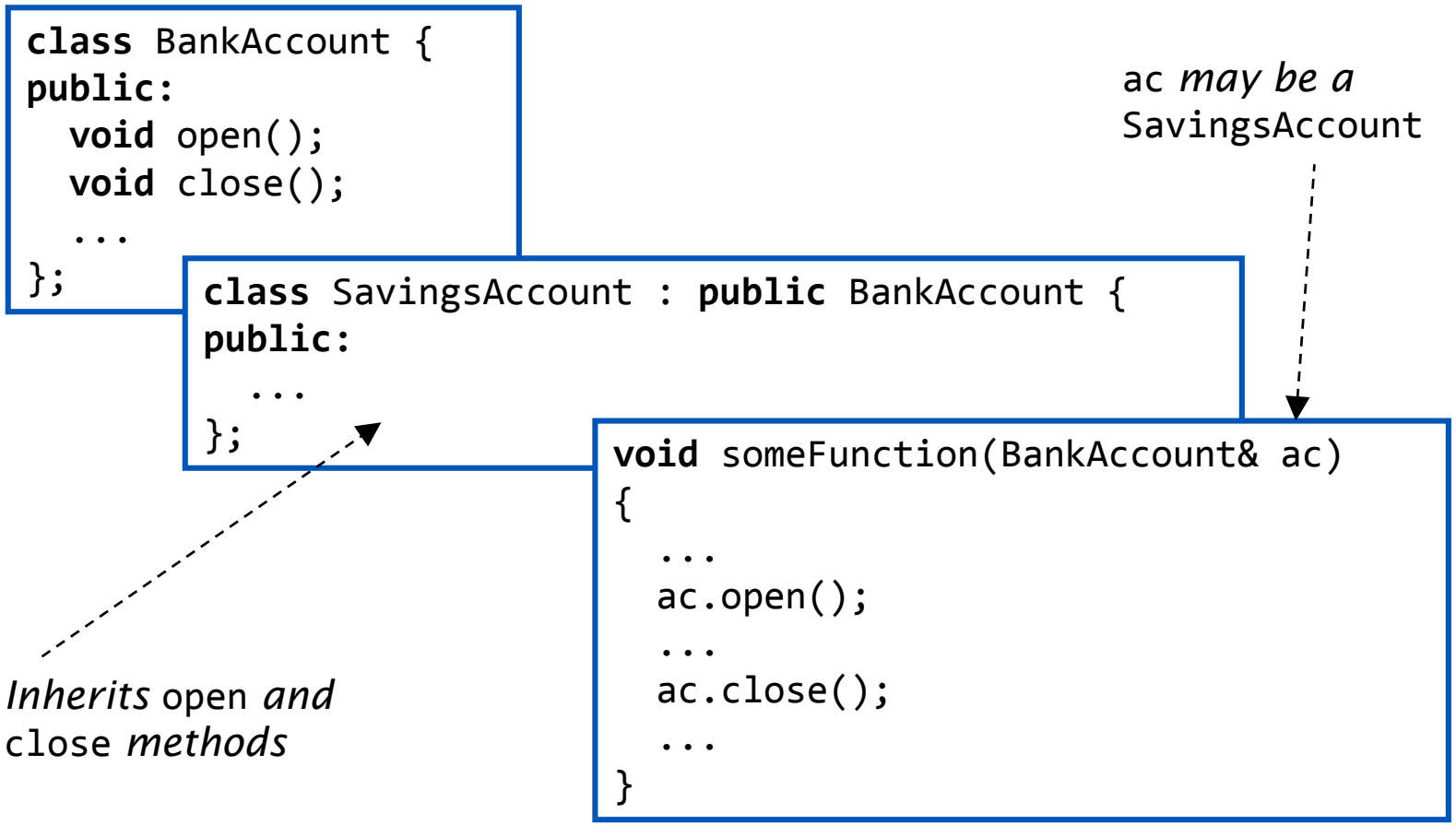
- Interface Inheritance and Polymorphism

- **Development Tools**



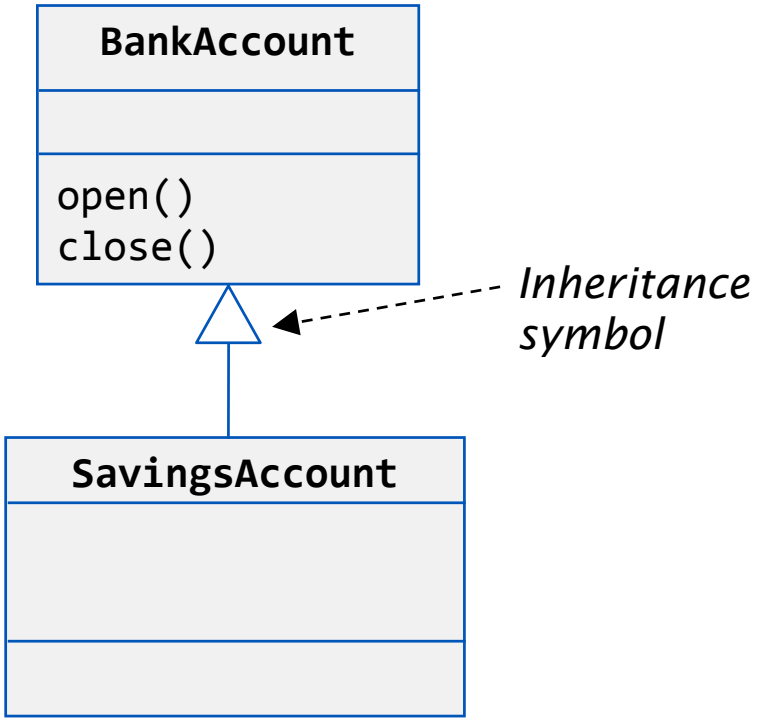
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Public Implementation Inheritance in C++



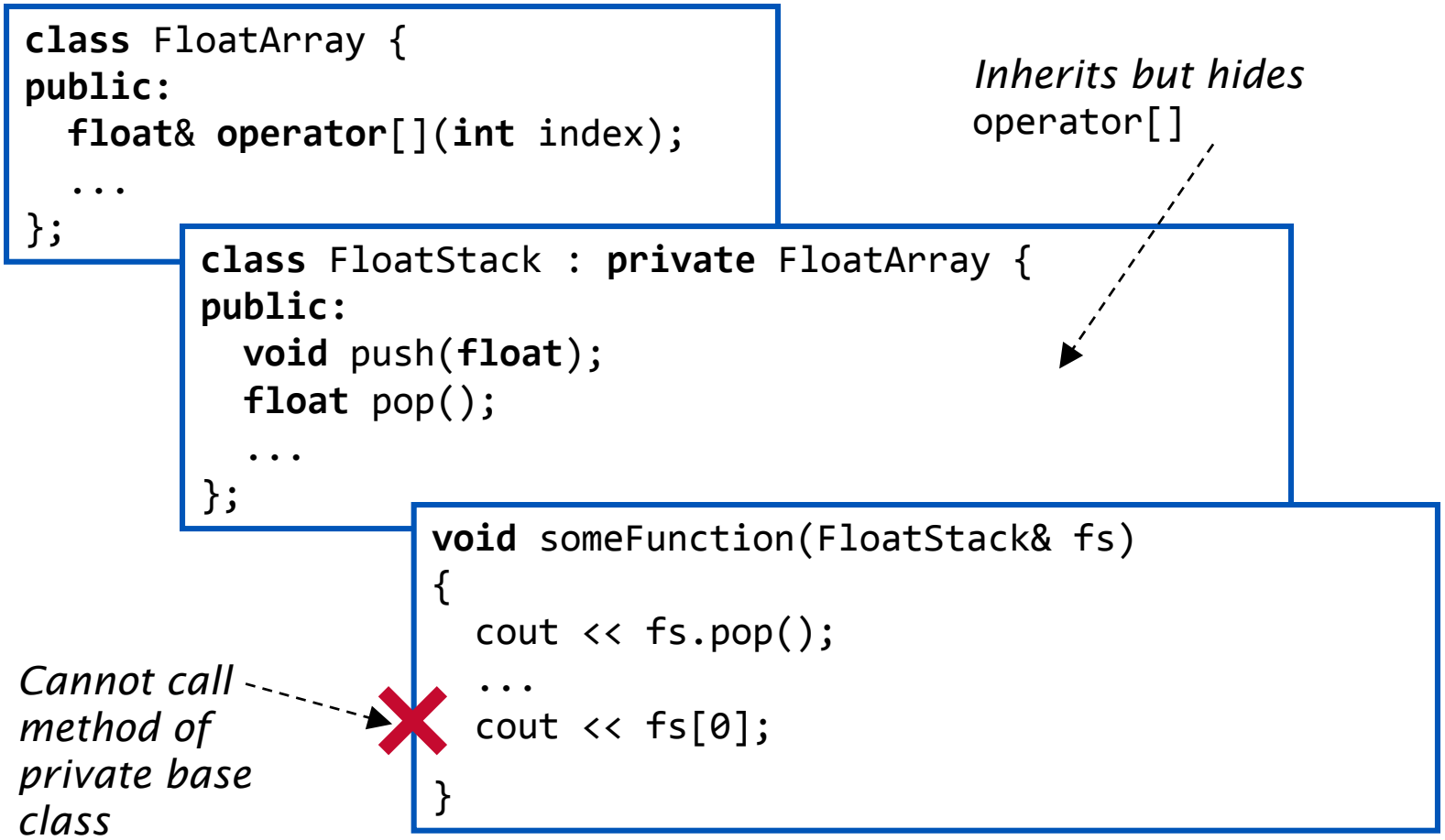
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Public Implementation Inheritance in UML



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

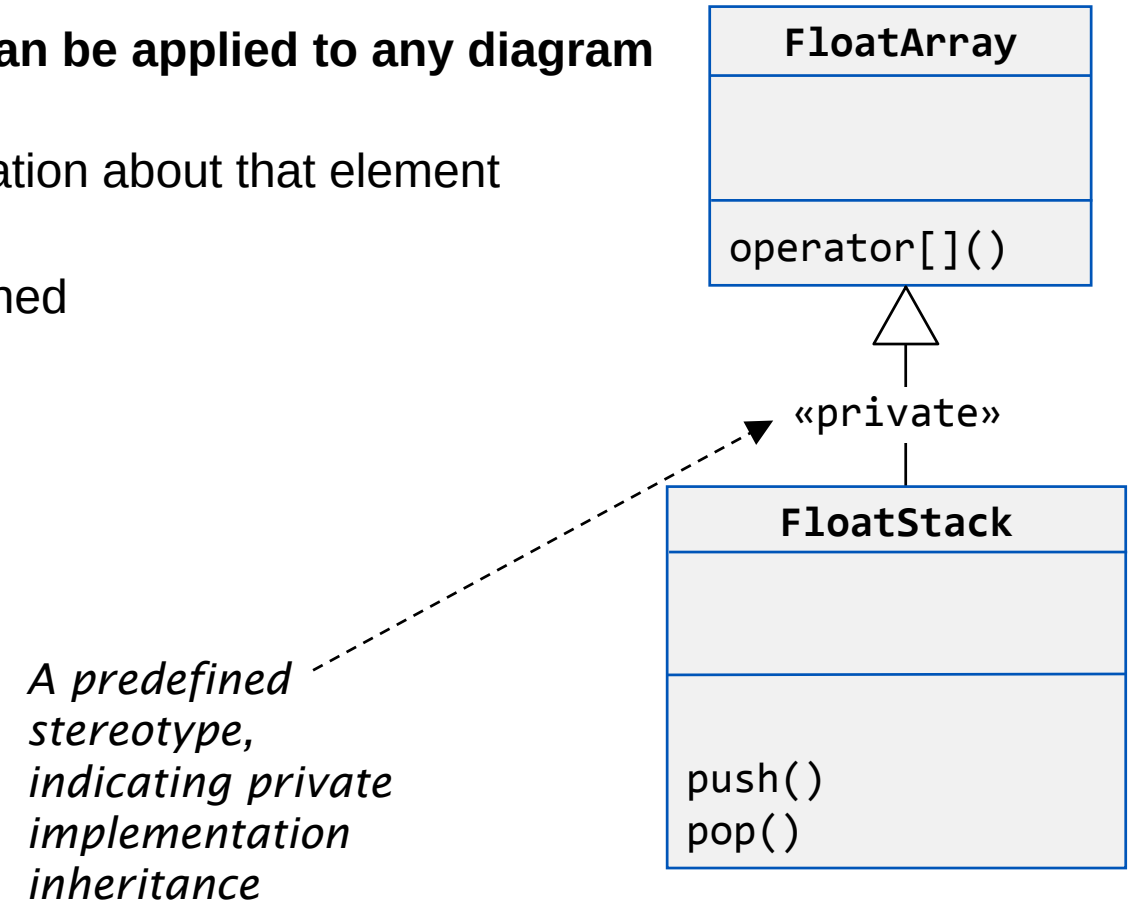
Private Implementation Inheritance in C++



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Private Implementation Inheritance in UML

- **UML has no specific notation for private inheritance**
- **However, *stereotypes* can be applied to any diagram element**
  - Provides more information about that element
  - Some are predefined
  - May also be user-defined



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

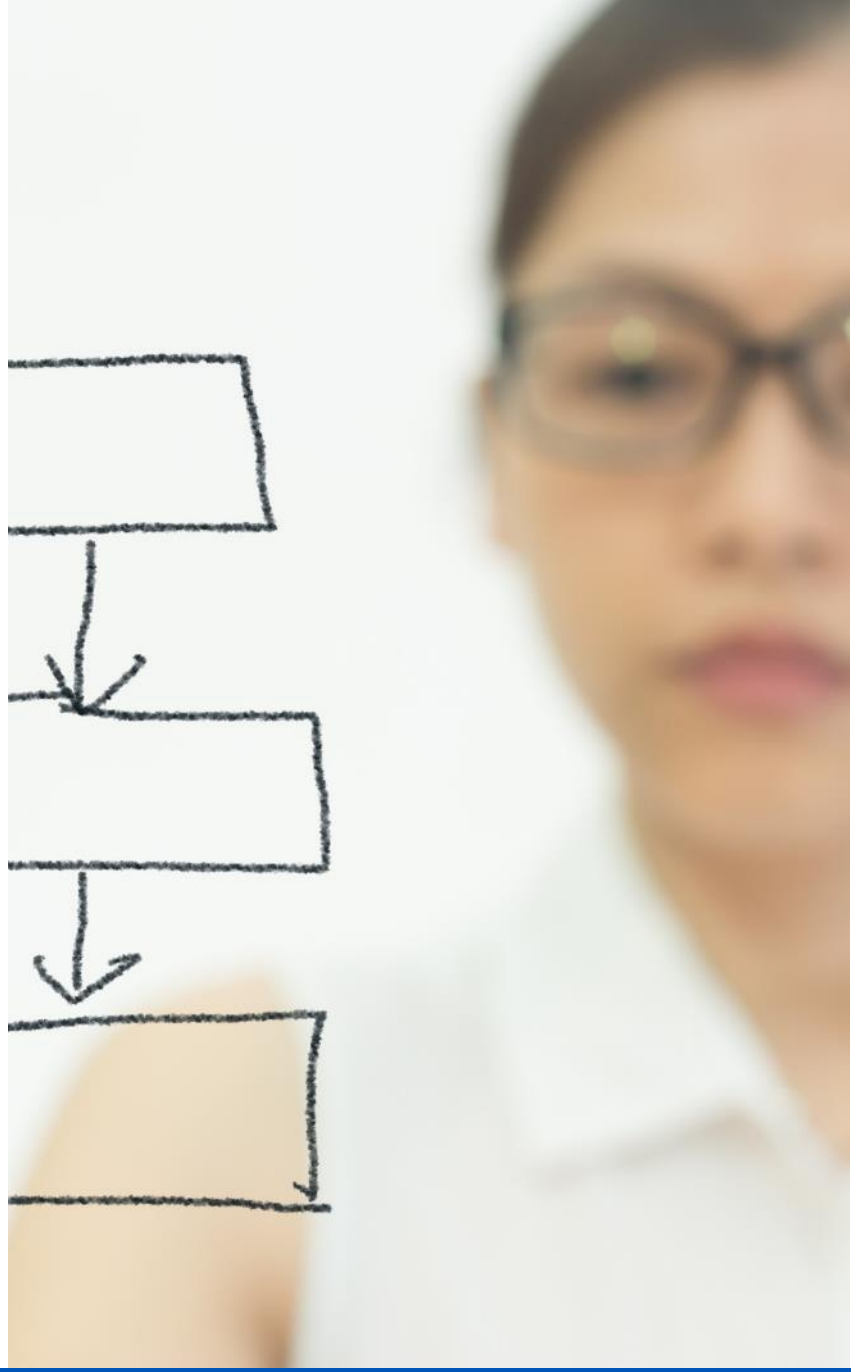
Contents

The OO Paradigm, UML, and C++

- Classes, Attributes, and Methods
- Objects, Messages, and Associations
- Aggregation
- Implementation Inheritance

Interface Inheritance and Polymorphism

- **Development Tools**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Interface Inheritance in C++

- **Strict interface inheritance is achieved by deriving from an *abstract base class* that has only *pure virtual* member functions**
  - No other member functions or data members
- **A derived class must implement those functions**
  - This is called *implementing* the interface

`I` *prefix is a common convention for interface names. (Some like it, some do not!)*

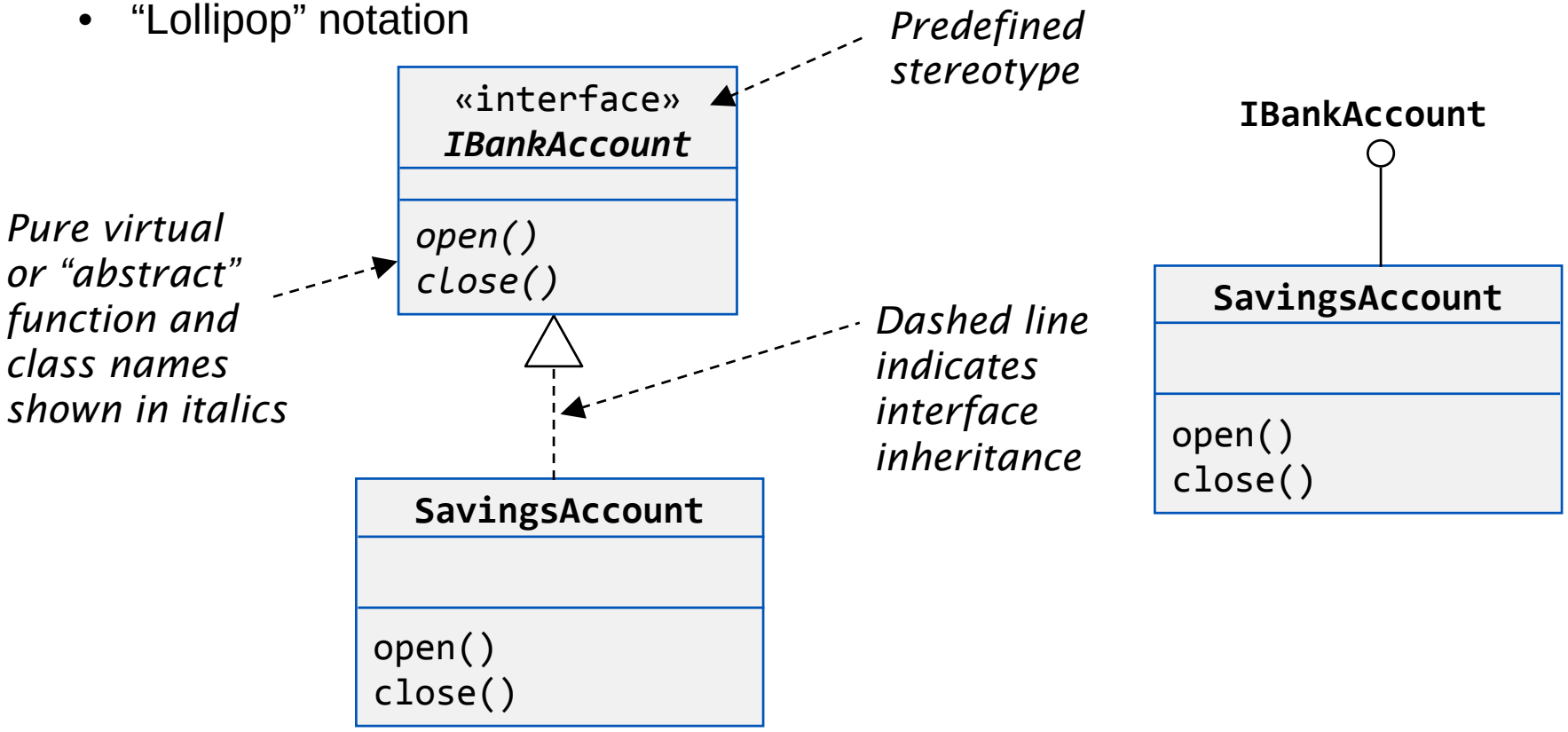
```
class IBankAccount {
public:
  virtual void open() = 0;
  virtual void close() = 0;
  virtual ~IBankAccount(){}
  ...
};
```

```
class SavingsAccount : public IBankAccount{
  ...
  void open(){ ... }
  void close(){ ... }
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Interface Inheritance in UML

- **Interface inheritance can be shown in either of two ways**
  - Stereotyped inheritance notation
  - "Lollipop" notation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Implementation vs. Interface Inheritance

- **These are separate concepts that are often confused in C++**
  - A C++ base class may mix pure virtual member functions with data members and member functions that have implementations
- **Many other languages treat interface inheritance very differently from implementation inheritance**
  - Java and C# allow multiple interface inheritance but only single implementation inheritance, and use the keyword **`interface`** for classes with nothing but pure virtual functions
  - Java also uses two separate keywords to indicate the two different ways of inheriting: **`extends`** vs. **`implements`**
- ***Polymorphism* is a feature of inheritance**
  - Each class that implements the interface provides a different implementation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

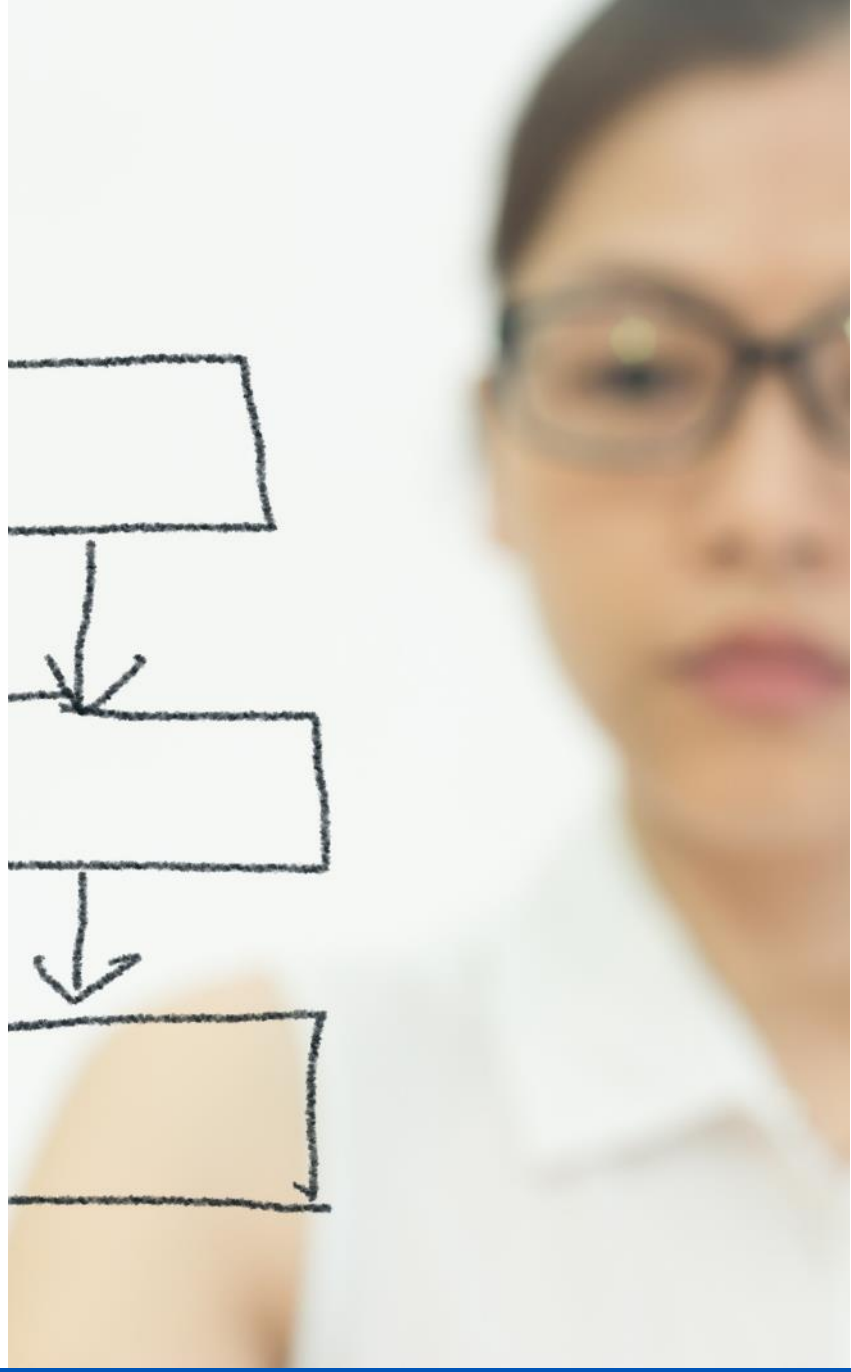
Contents

- **The OO Paradigm, UML, and C++**

Development Tools

Development Tasks

- CASE Tools
- Integrated Development Environments
- Source Control



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

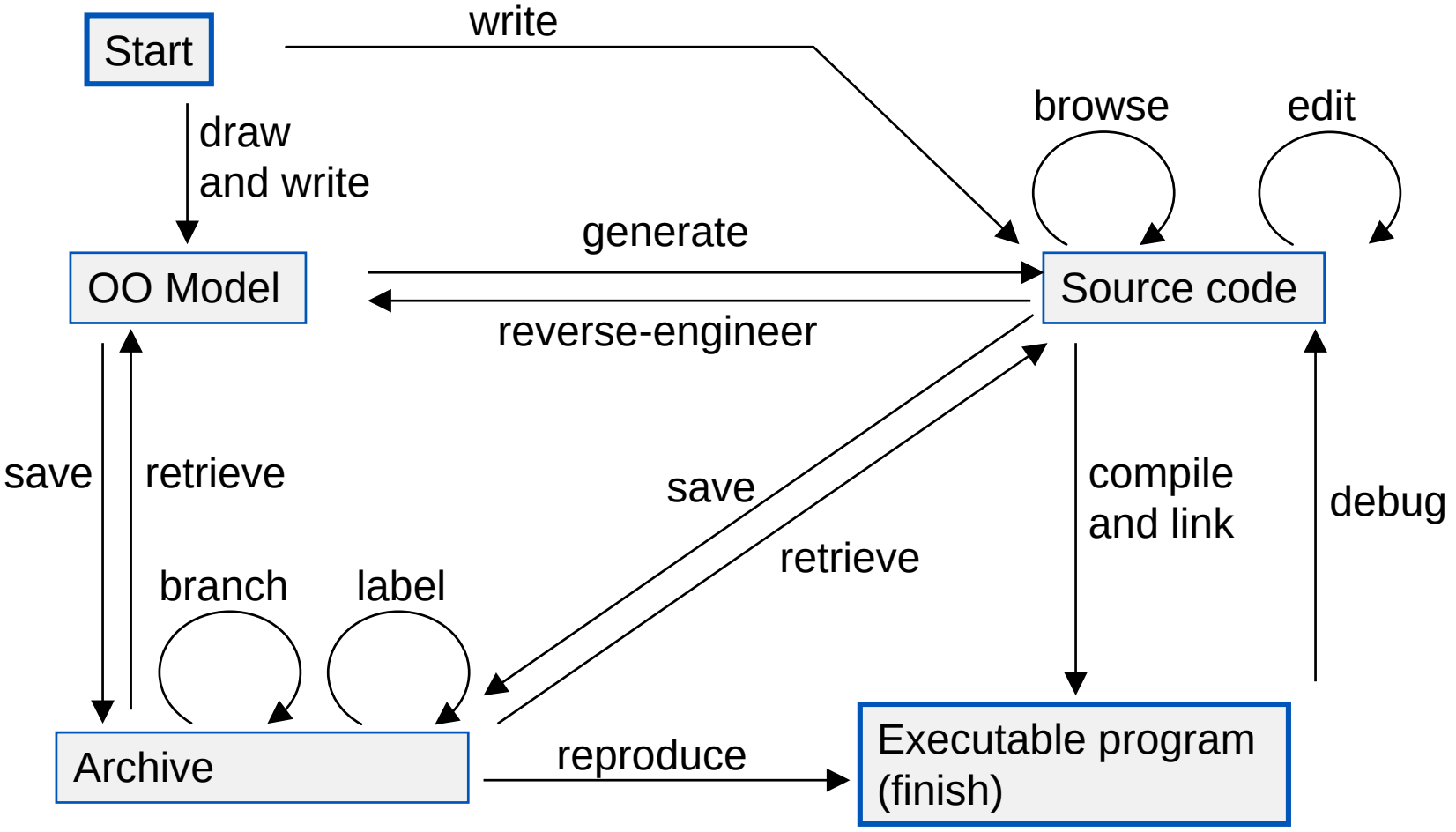
Development Tasks

- **This section discusses the tasks that face most C++ developers and the tools that help support them**
- **We take the developer's job as starting with some sort of specification and ending with a working, executable program**
  - We also recognize maintenance tasks that form a large part of the through-life cost of most software
- **The diagram on the following slide summarizes the most common development tasks**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Summary of Development Tasks



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

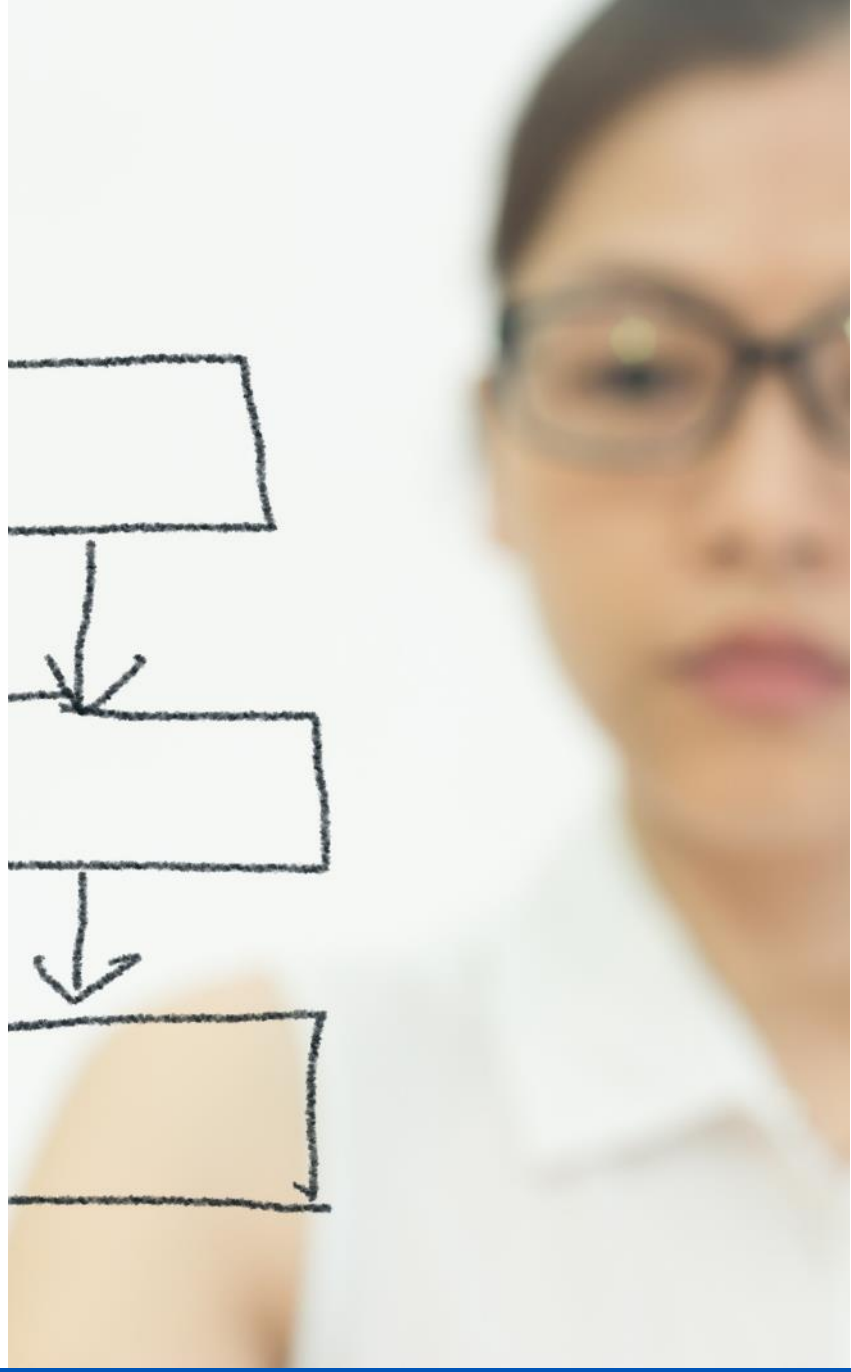
- **The OO Paradigm, UML, and C++**

Development Tools

- Development Tasks

CASE Tools

- Integrated Development Environments
- Source Control



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

CASE Tools

- **Computer-Aided Software Engineering (CASE) tools generally provide**
  - A drawing tool supporting the diagrams required by a particular modeling notation
  - A *data dictionary* or *repository*
  - Consistency checking between diagrams and the repository
  - Report or document generation
- **A CASE tool may also provide**
  - Code generation, including database schema generation
  - A code browser
  - Group working facilities, such as locking of diagrams
  - Version control
  - Reverse engineering of models from existing code
  - Assistance in using the methodology it supports

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

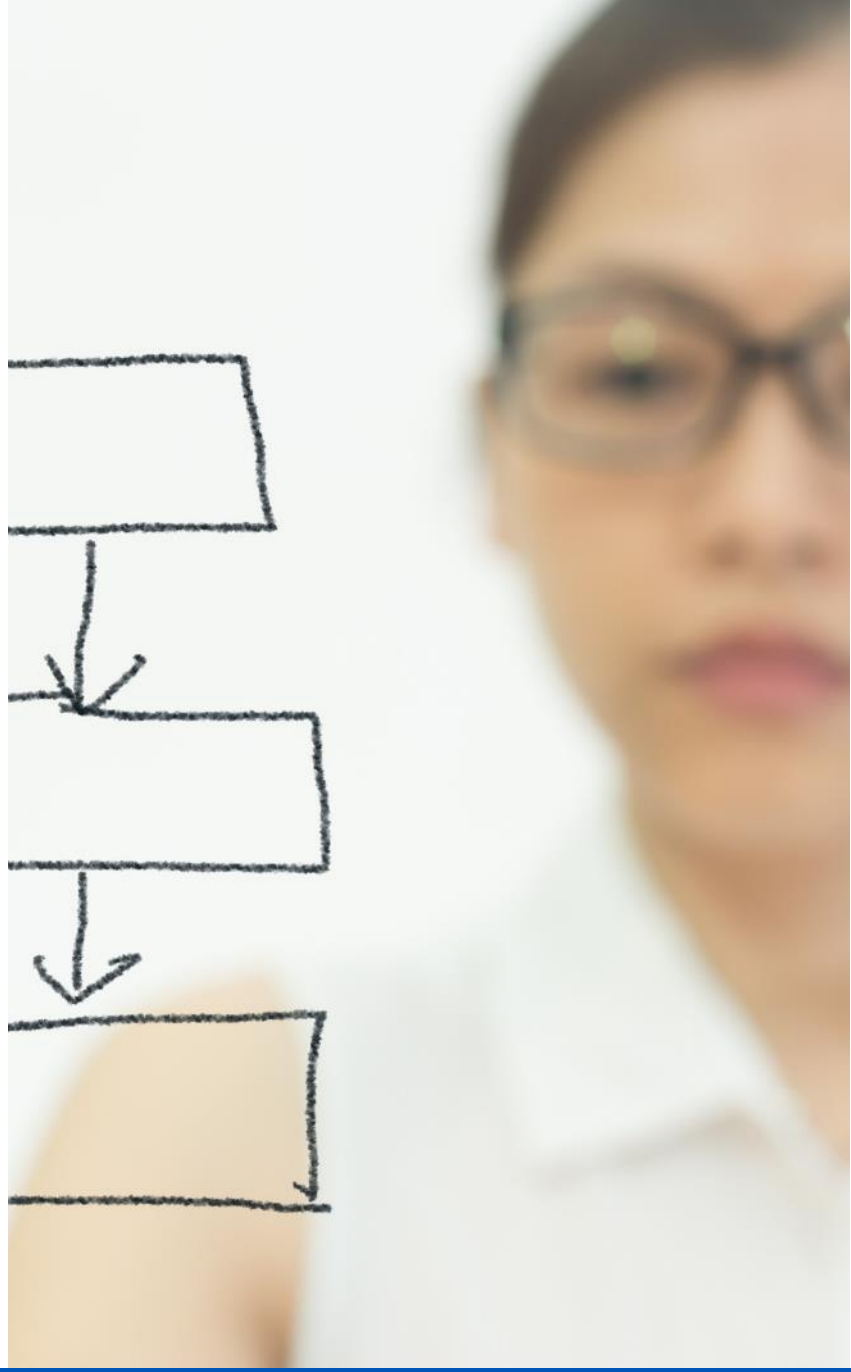
- **The OO Paradigm, UML, and C++**

Development Tools

- Development Tasks
- CASE Tools

Integrated Development Environments

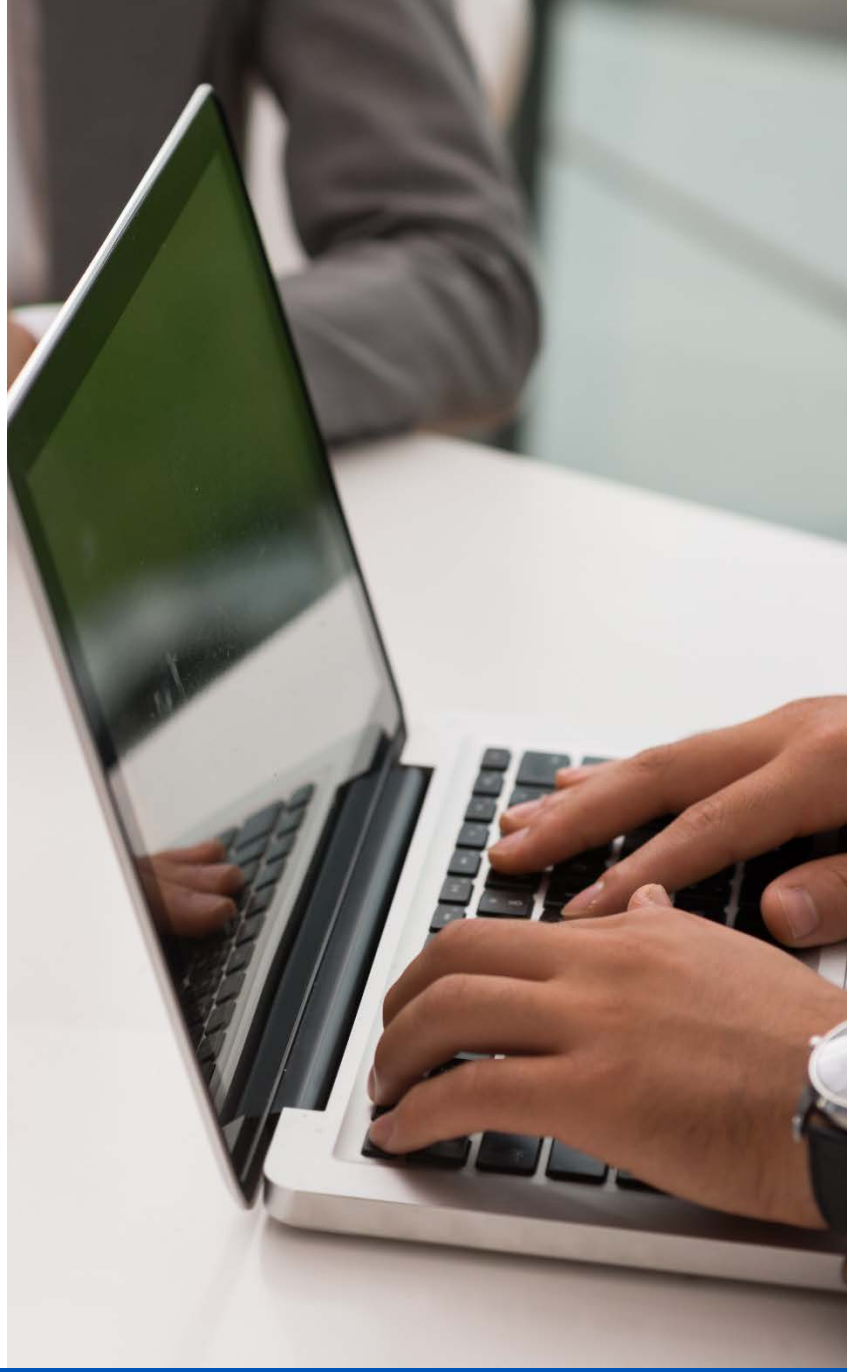
- Source Control



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Integrated Development Environment

- **An Integrated Development Environment (IDE) allows the developer to perform most development tasks from within a single environment**
- **Microsoft Visual Studio provides**
  - Context-sensitive editor
  - Class browser
  - Compiler and linker with “make” facility
  - Source-level debugger
  - Refactoring help
  - Etc.



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

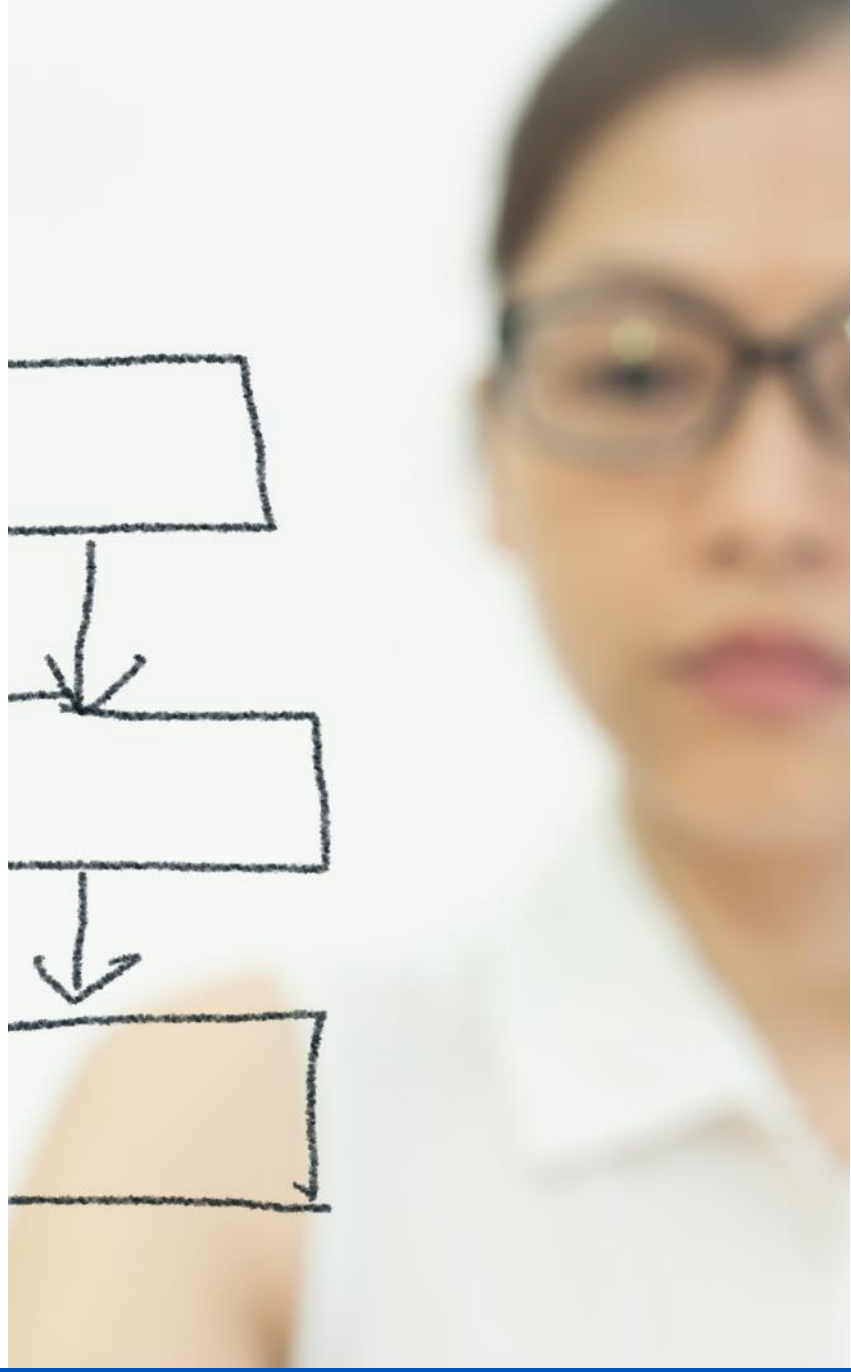
Contents

- **The OO Paradigm, UML, and C++**

Development Tools

- Development Tasks
- CASE Tools
- Integrated Development Environments

Source Control



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Source Control

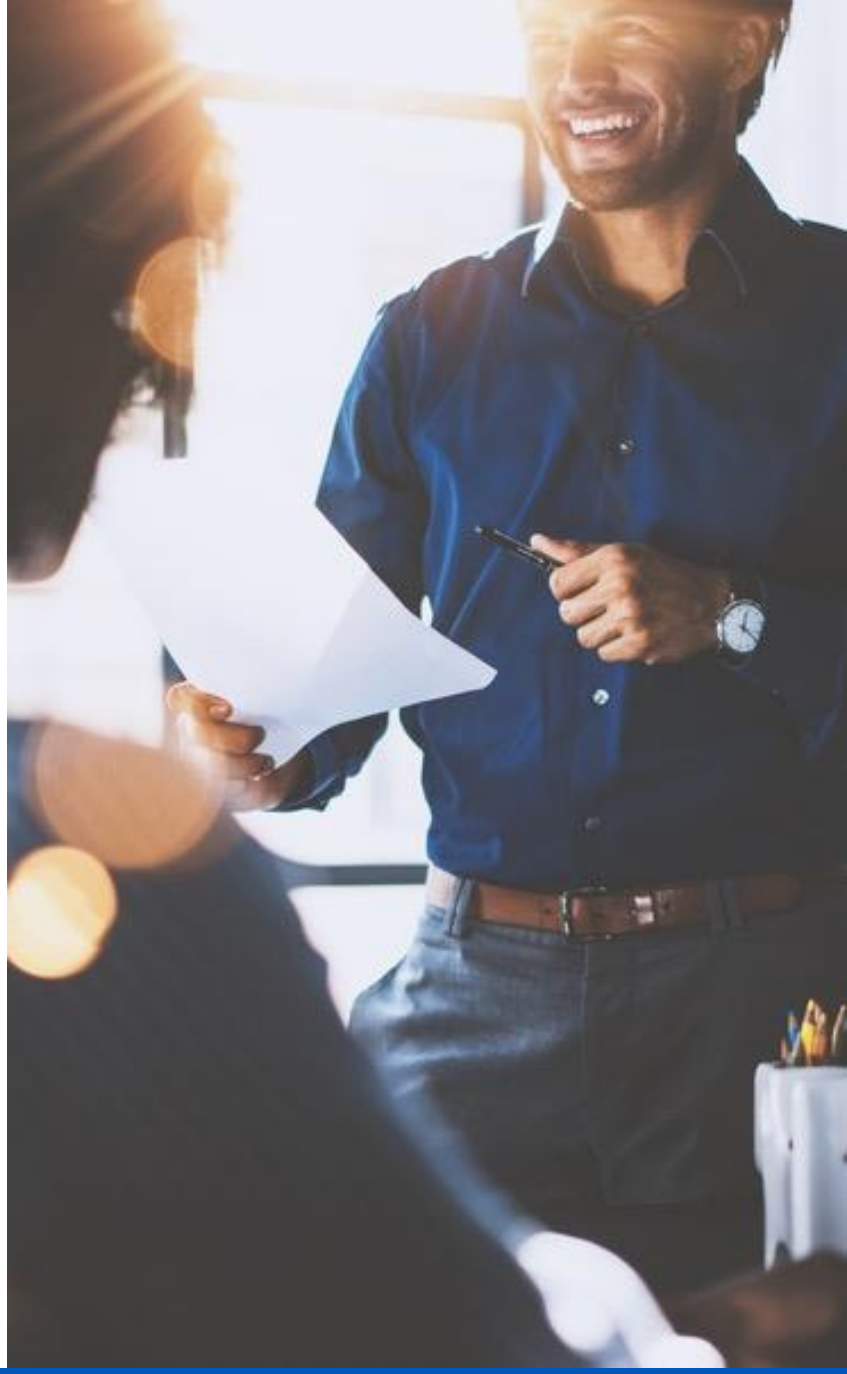


- **Source control is a key part of any large development**
- **Source-control features include**
  - Version control of individual files
  - Workgroup coordination
  - Labeling of files that constitute a coherent build
  - Branching to allow concurrent, independent development paths

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- **Learn the concepts underlying OO design and programming**
  - Using UML to annotate OO designs
  - How C++ implements these concepts
- **Learn how to take advantage of modern tools for C++ development**
  - Visual Studio



OO = object-oriented
UML = Unified Modeling Language

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



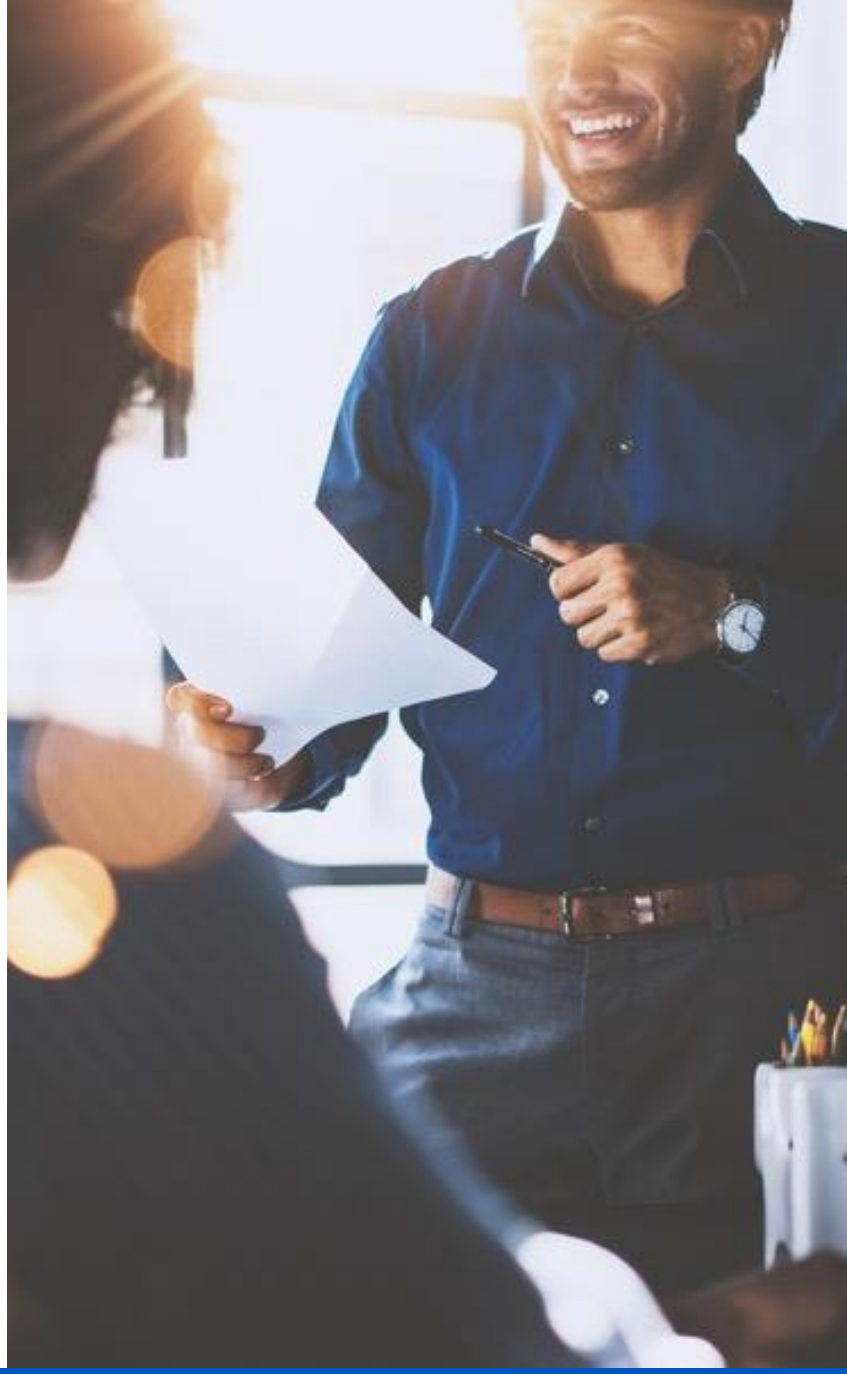
Chapter 2

Patterns, Part 1: Design Patterns

LEARNING TREE INTERNATIONAL

Objectives

- **Learn what a *design pattern* is**
- **Describe design patterns**
- **Work with three general design patterns**
  - Adapter
  - Singleton
  - Template Method



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Design Patterns

Describing Design Patterns

- The Singleton Pattern
- The Template Method Pattern



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Pattern Example: Adapter

- **Problem: Usage of a service is desired, but the interface is not the one you expect/want**
  - For example, cables: foreign power plugs, USB plug variants, etc.
  - Solved by using a physical adapter



- **The same problem occurs in software**
  - Code has been written to use a service; e.g., a library
  - We want to include a new external library to extend our service
    - The new code does the job, but has a different interface
    - It is neither feasible to rewrite our own code nor the new library
  - Solution?

- **How do we best describe the idea of an Adapter and discuss all of its aspects?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Patterns

- **OO provides a language for describing systems and software**
- **Classes, attributes, methods, inheritance, etc., are the "words" of this language**
  - All possible OO solutions can be built from these words
- **Design patterns provide a higher-level abstraction for describing problems and their solutions**
  - Equivalent to commonly understood concepts, phrases, or sentences
- **Much easier to communicate using known concepts and phrases than starting from individual words**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Patterns in Other Disciplines

- **Practitioners of many disciplines use patterns to communicate accepted ideas efficiently**
  - Here are some examples:

Discipline	Low-level language	High-level or pattern language
Music	Stave notation	12-bar blues, plagal cadence, sonata form
Poetry	Description of rhyming and scan	Limerick, haiku, iambic pentameter
Electronics	Component-level circuit diagrams	LCR network, potential divider
Your favorite discipline		

LCR = Inductor-Capacitor-Resistor

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Pattern: Name

- **A pattern needs a good name**
  - Should describe the intent, not only the structure
    - “Wrapper,” “Callback,” or “Model 2” are not good names
    - “Adapter” and “Observer” are better
- **Good names will enhance the level of design communication and discussion and make them more efficient**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Pattern: Solution to a Recurring Problem

- **To be a pattern, it needs to solve a recurring problem, not a once-off**
- **The solution is not a final one, but a very good start**
  - It is not meant to be a plug-in, ready-made, stop-thinking solution
  - A design pattern provides an advanced starting point
    - Do not reinvent the wheel
    - Do not wander into quagmires where others have already been struggling
- **A design pattern should provide easy-to-use, pedagogic, and well-packaged expert knowledge and experience**
  - Can you find an argument against that?
- **Each design pattern tends to become a research topic**
  - A structure for continuous improvement

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Pattern: Context

- **A design pattern describes a solution in a certain context**
- **Just because a design pattern exists, it should not be applied just anywhere**
  - A pattern is not good by nature
- ***Suspension bridge* and *beam bridge* are both well-known bridge design patterns**
  - Why don’t we build suspension bridges everywhere?
  - Why don’t we build beam bridges everywhere?

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Describing Design Patterns

- **The de facto standard for describing design patterns is the book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides**
  - Published in 1995
  - Known as the Gang of Four (GoF)
- **Each pattern is described in terms of these elements**
  - Identification
    - Name
    - Classification
    - Intent
  - Problem
    - Motivation
    - Applicability
    - Known uses
  - Solution
    - Structure
    - Participants
    - Collaborations
  - Consequences
    - Implementation
    - Sample code
    - Related patterns

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Design Patterns

- Describing Design Patterns

The Singleton Pattern

- The Template Method Pattern



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Identification

- **Name**
  - Alternative, or "*also known as*," names may also be given
- **Classification**
  - Purpose—may be one of:
    - *Creational*: concerned with object creation
    - *Structural*: deals with object or class composition
    - *Behavioral*: deals with object interaction
  - Scope—may be one of:
    - *Class*: deals with relationships between classes
    - *Object*: deals with objects
- **Intent**
  - Indicates the design issue or problem addressed by the pattern

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton: Identification

- **Name**
  - Singleton
- **Classification**
  - Creational
  - Object
    - Manages the creation of objects
- **Intent**
  - Ensure that a maximum of *one* instance of a class will exist, and provide a global point of access to it

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Problem

- **Motivation**
  - Illustrates the design problem using a realistic example scenario
  - Shows how the pattern solves the problem
- **Applicability**
  - The situations in which the pattern can be applied
  - Examples of poor design that the pattern addresses
  - How to recognize these situations
- **Known uses**
  - Examples of the pattern found in real systems

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton: Problem

- **Motivation**
  - It is necessary for some classes to have only one instance; e.g., to hold systemwide state information or manage a resource
  - A global variable makes an object accessible, but has drawbacks
    - Unwieldy: What should it be called? Where should it be declared?
    - Unreliable: It does not prevent the declaration of other instances
- **Applicability**
  - Whenever there must be exactly one instance of a class (or none if not used)
- **Known uses**
  - Ubiquitous

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

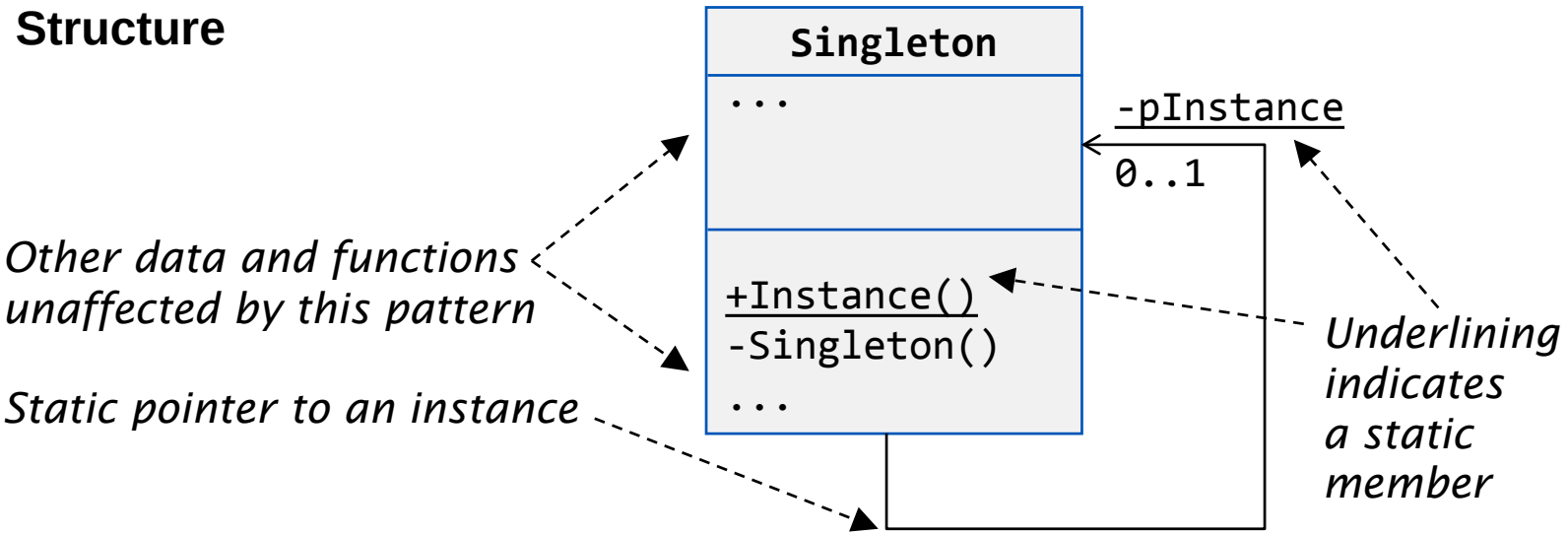
Solution

- **Structure**
  - A graphical representation of the classes in the pattern
  - We use UML notation
- **Participants**
  - A description of the classes and/or objects that participate in the pattern
- **Collaborations**
  - Describes the collaborations or interactions between the participants

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton: Solution

- **Structure**



- **Participants**
  - A Singleton class defines a static member function that returns a pointer to the sole instance of the class
  - This `Instance` function creates the instance if it does not already exist
- **Collaborations**
  - Clients access the instance solely through this function

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Consequences

- **Consequences**
  - Benefits and drawbacks of using the pattern
- **Implementation**
  - Pitfalls, hints, and techniques to be aware of when implementing the pattern
  - Language-specific issues
- **Sample code**
  - Code fragments that illustrate how the pattern might be implemented in a variety of languages
    - We will show only C++
- **Related patterns**
  - Similar patterns, differences being highlighted
  - Other patterns with which this one can be used

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton: Sample Code—GoF Singleton

```
// SysState.h
class SysState {
  ...
public:
  static SysState* Instance();
private:
  static SysState* pInstance;
  SysState();
};
```

```
// SysState.cpp
#include "SysState.h"
...
SysState* SysState::Instance(){
  if(pInstance == nullptr){
    pInstance = new SysState;
  }
  return pInstance;
}

SysState* SysState::pInstance = nullptr;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton: Sample Code—GoF Singleton

Keeping a local pointer to the instance will compile, but is bad practice; `SysState` *class may choose to destroy the current instance and create a new one, making pointer invalid*

Compilation error, constructor is private

The preferred way to use a singleton

```
#include "SysState.h"

SysState* myPrivatePtr = SysState::Instance();

int main()
{
  SysState myPrivateState;

  ...

  SysState::Instance()->someFunc();

  ...
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code—Meyers' Singleton

- **In the GoF singleton, the instance is declared at class scope**
- **In *Effective C++*, Meyers declares it in the `Instance()` function**
  - As a static local variable
- **When is the `instance` initialized and destroyed?**

```
class SysState {
  ...
public:
  static SysState& Instance();
private:
  SysState();
};
```

```
#include "SysState.h"
...
SysState& SysState::Instance(){
  static SysState instance;

  return instance;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code—Nifty Counter

- **The *nifty counter* implementation ensures the instance is initialized before it is needed and cleaned up after its last possible use**
  - It guarantees that the singleton gets destroyed safely
- **How does this work? Is it too smart?**

```
class SysState {
...
  static SysState* Instance();
public:
  class SSCounter {
    static int count;
  public:
    SSCounter();
    ~SSCounter();
  };
};

static SysState::
   SSCounter counter;
```

```
...
SysState::SSCounter::SSCounter(){
  if(++count == 1){
    SysState::pInstance = new SysState;
  }

SysState::SSCounter::~SSCounter(){
  if(--count == 0){
    delete SysState::pInstance;
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton: Consequences

- **Consequences**
  - Reduced name space; no need to pollute the global name space with names of sole instances
  - The alternative approach of declaring a global instance does not prevent programmers from declaring more instances
  - The alternative of using all **`static`** methods is also flawed
    - **`static`** methods cannot be virtual
    - All data members must also be **`static`**, and the initialization order between compilation units is undefined
- **Implementation**
  - Constructors should be made private or protected in C++ to ensure that instances cannot be created by clients
    - **`protected`** allows instances of derived classes to be created
    - **`private`** prevents this
  - The copy constructor, assignment operator, and destructor should be declared **`private`** to prevent copies, or removed with **`=delete`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aside: C++11 `default/delete`

- **One issue in traditional C++ is that you automatically get a default and copy constructor, as well as an assignment operator**
  - These are quite often *not* the ones you want
  - To hide them, you would traditionally declare them **`private`**
  - C++11 has added syntax to remove them completely
    - Or explicitly say that you want the default implementation as before
    - Best practice: “As you can see, I have explicitly thought about constructors”

```
class SysState {
        ...

        SysState () = default;
        SysState (const SysState &) = delete;
        SysState& operator=(const SysState &) = delete;

        ...
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

C++ Keyword `explicit`

- **By default, a constructor with a *single* parameter will act as an implicit conversion operation, applied automatically**
  - This is sometimes convenient, but often absolutely not what you want
  - Use **`explicit`** on such a constructor to prevent the implicit conversion

```
class Thing{
    private:
        int size;
        ...
    public:
        explicit Thing(int s): size(s){
            ...
        }
        ...
};
...
Thing t(8);
...
t = 56;
```

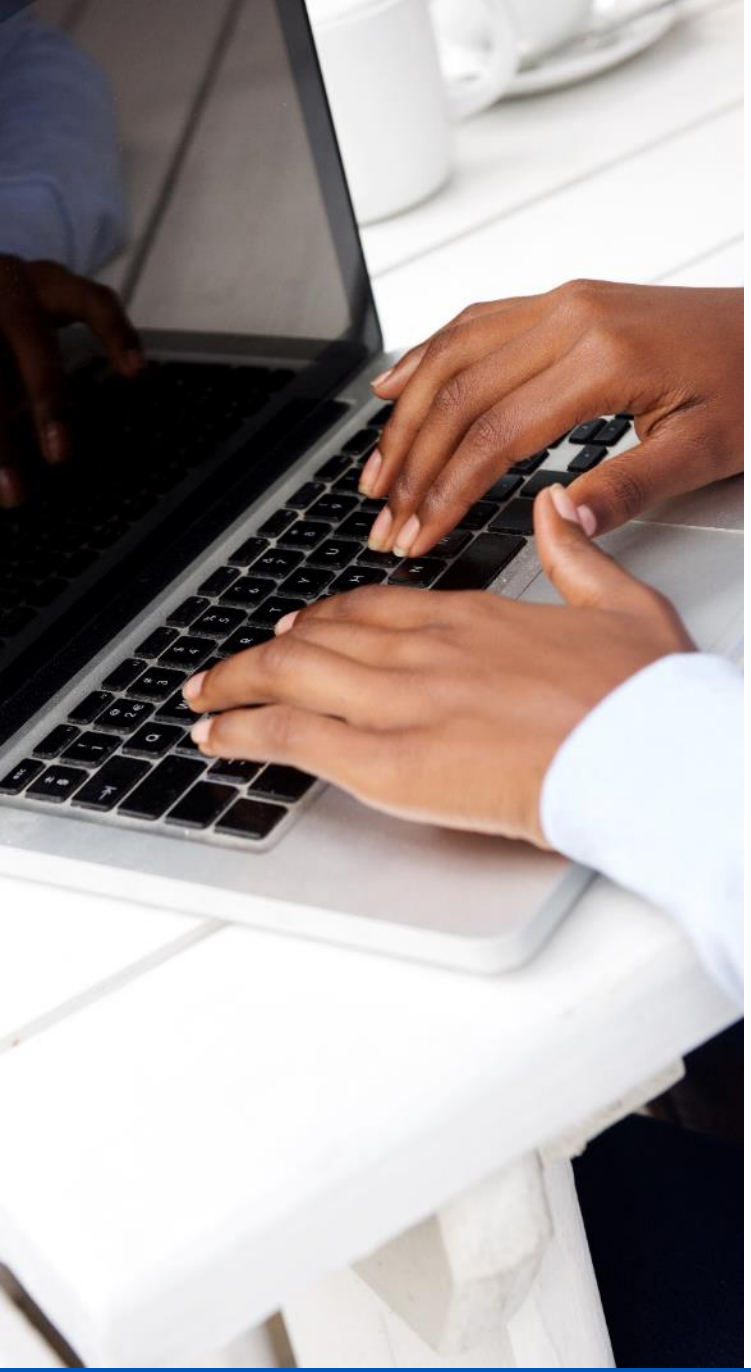
Without **`explicit`**, `56` *would be automatically converted to a new* `Thing`, *which would then be assigned to* `t`, *replacing its object value*

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton: Summary

- Singleton feels like a fairly easy pattern, yes?
- How many of the pitfalls would you have fallen into, not having read the Consequences and Implementation sections?

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 2.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 2.1: Singleton Pattern

In this exercise, you will be working on the pelican crossing application

You will

- Spot the error in the use of the `TimerSet` class that prevents the lights from cycling
- Fix the error by converting the `TimerSet` class into a singleton and using it correctly

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Design Patterns

- ► Describing Design Patterns
- ► The Singleton Pattern

The Template Method Pattern



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Identification

- **Name**
  - Template Method
- **Classification**
  - Purpose: Behavioral
  - Scope: Object
- **Intent**
  - Define the skeleton of an algorithm in a base-class member function, deferring some steps to derived classes
  - Template Method lets derived classes define certain steps of an algorithm without changing the algorithm's structure

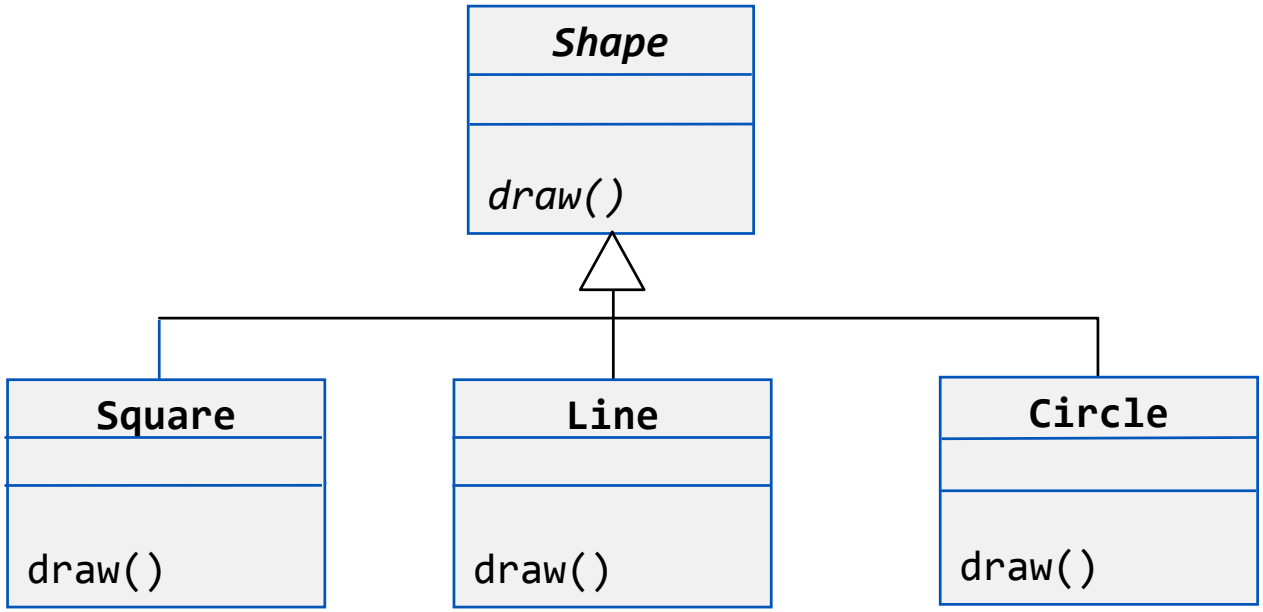


The Template Method pattern should not be confused with C++ templates

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Motivation

- **Conventional use of polymorphism**
  - A family of derived classes may need to provide specialized redefinitions of a base-class operation
  - Conventionally, the base class declares a virtual member function that derived classes then implement
- **Example: drawing shapes in a graphics package**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Motivation

- **Using this approach, each derived class sometimes must implement the common structure of the operation**
  - Suppose each `draw` operation must lock the graphics resource before drawing, and unlock it afterward

```
void Circle::draw(){
  lock();
  // draw the circle

  unlock();
}
```

```
void Square::draw(){
  lock();
  // draw the square

  unlock();
}
```

- **It would be better to implement the common structure of the operation only once**
  - This is what the Template Method pattern achieves

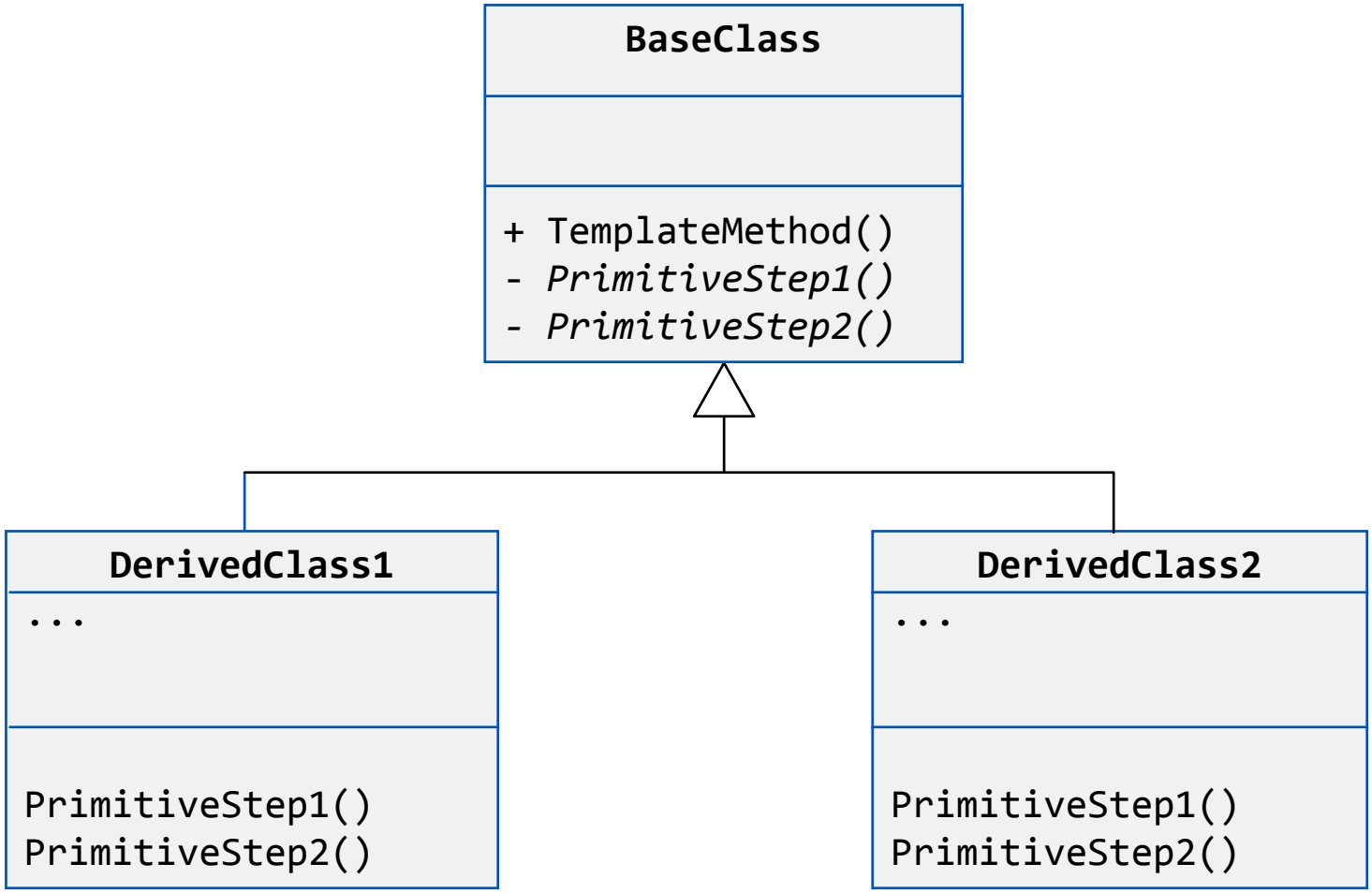
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Applicability

- **Applicability**
  - The Template Method pattern is used whenever a family of derived classes needs to implement an algorithm that has a common structure
  - The common structure is implemented in the base class
  - The derived classes implement only the variant parts
- **Known uses**
  - Just about everywhere

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Structure



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Participants and Collaborations

- **Participants**
  - Base class (e.g., `Shape`)
    - Declares the primitive steps the derived classes must implement
    - Implements the algorithm skeleton in a Template Method
  - Derived classes implement the primitive steps
- **Collaborations**
  - Clients invoke the base-class method, which then calls down the hierarchy to the derived-class implementations of the variant parts
  - This inverted control structure is sometimes called the *Hollywood principle:* "Don't call us; we'll call you"

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Sample Code

```
class Shape{
public:
  void draw();
private:
  virtual void doDraw() = 0;
  ...
};
```

```
...
void Shape::draw(){
  lock();
  doDraw();
  unlock();
}
```

```
...
class Circle: public Shape{
private:
  void doDraw();
  ...
};
```

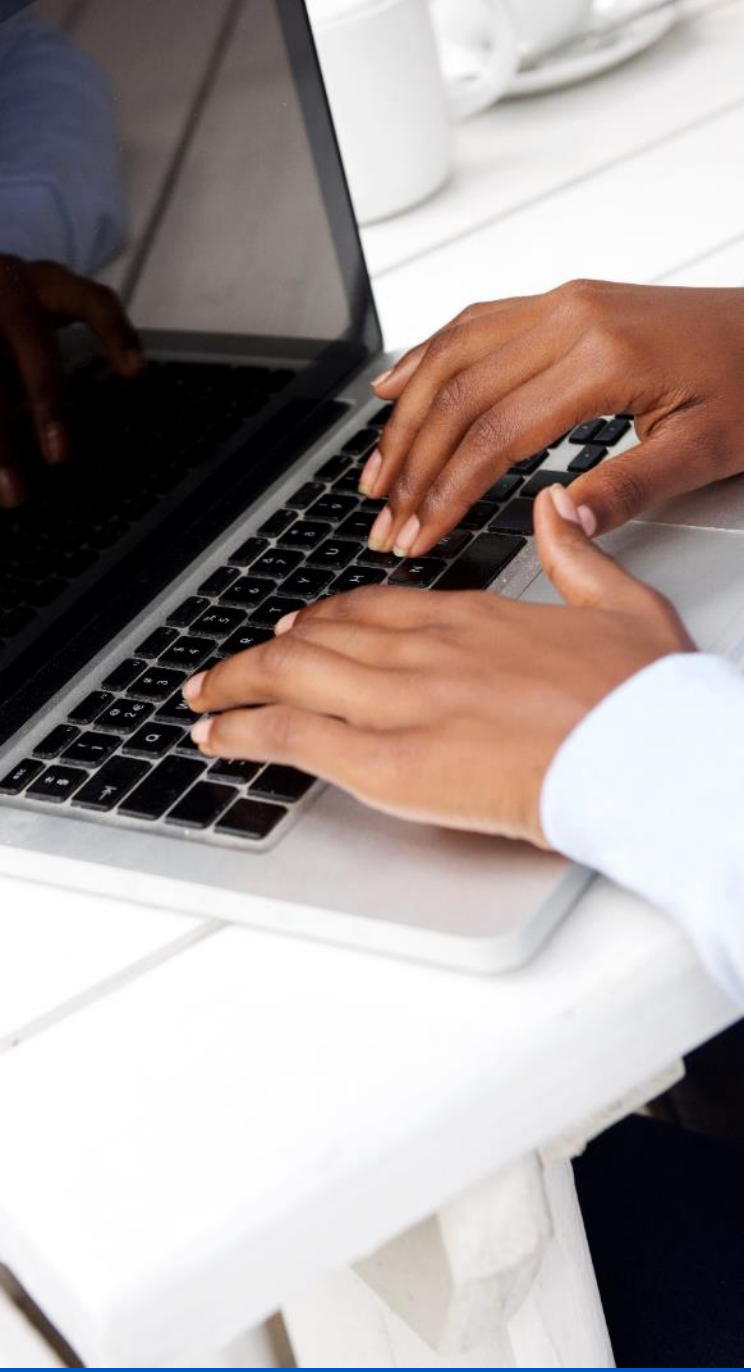
```
...
void Circle::doDraw(){
  // draw the circle
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method: Consequences

- **Consequences**
  - The Template Method pattern is a fundamental technique for code reuse
  - It allows a factoring out of common behavior into a base class without relying on derived classes to invoke it
- **Implementation in C++**
  - Primitive step functions can be made `protected` or `private` to ensure they are not called by clients
  - Primitive steps may be declared as
    - *Pure virtual*: Derived class must provide an implementation
    - *Simply virtual*: Base class provides a default implementation that derived classes may override
- **What could be a default `doDraw()` implementation in the previous example?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 2.2

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 2.2: Template Method Pattern

In this exercise, you will be working on the pelican crossing application

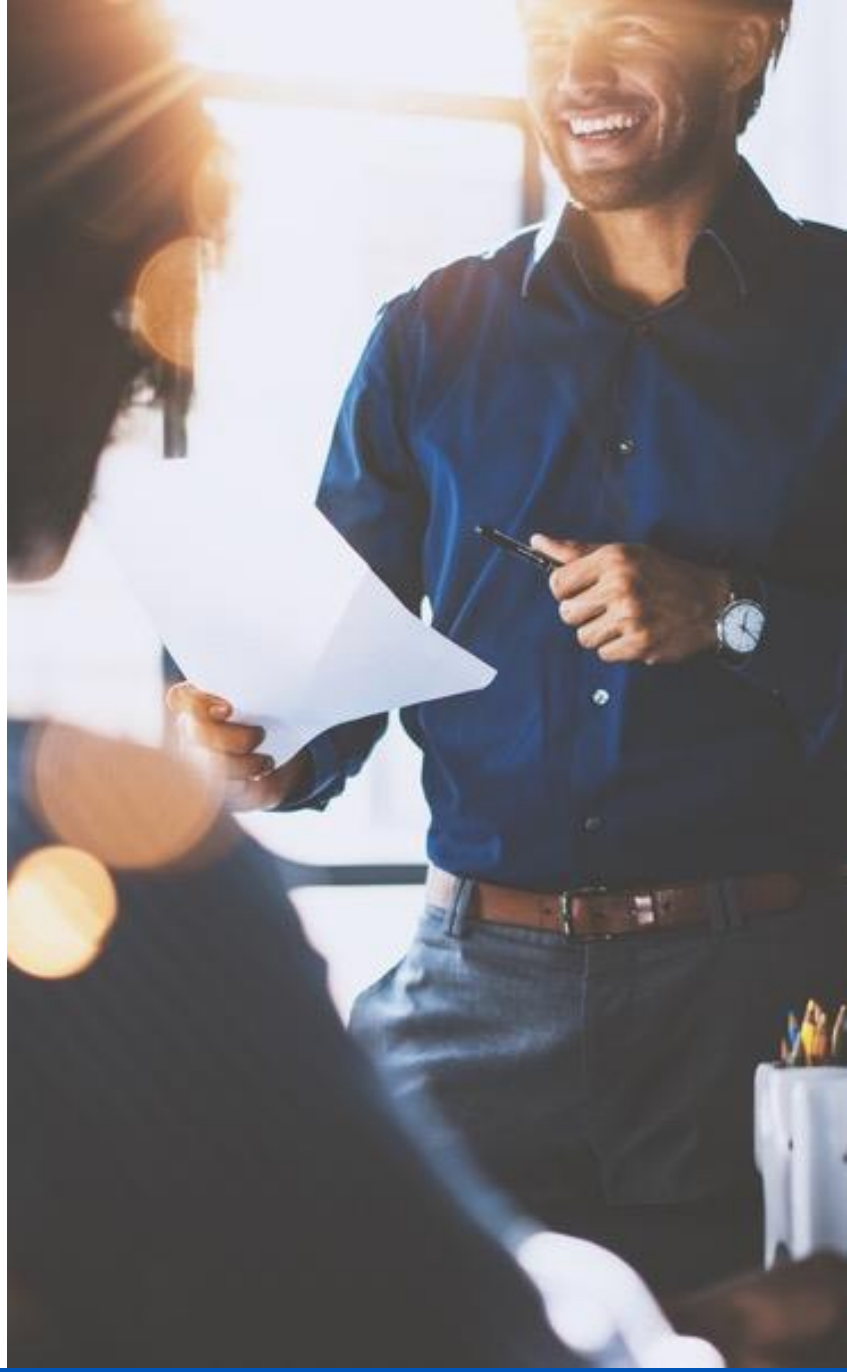
You will

- Convert the `Timer` class to make use of the Template Method pattern
- Modify the `PelicanTimer` class accordingly

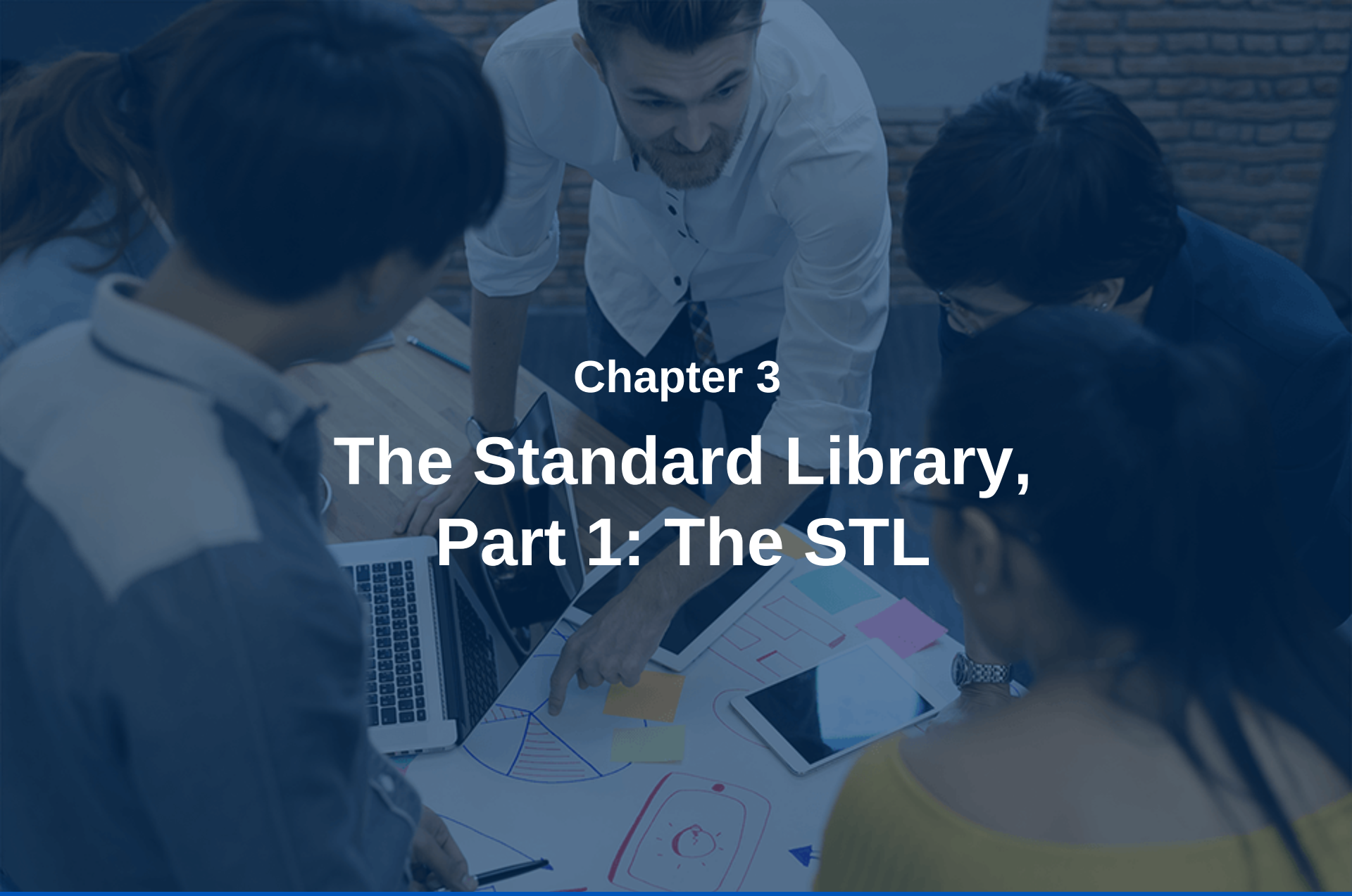
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- **Learn what a *design pattern* is**
- **Describe design patterns**
- **Work with three general design patterns**
  - Adapter
  - Singleton
  - Template Method



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



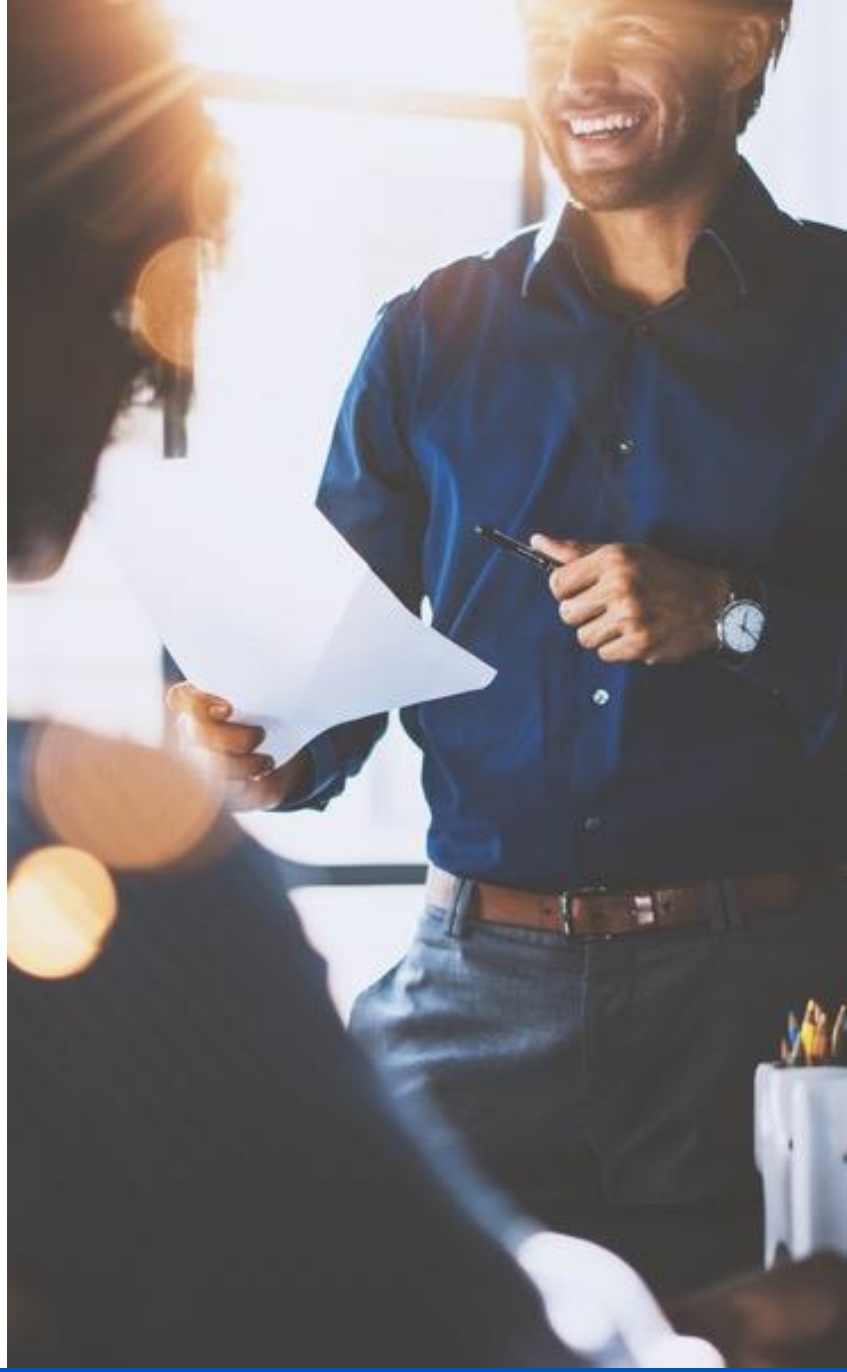
Chapter 3

The Standard Library, Part 1: The STL

LEARNING TREE INTERNATIONAL

Objectives

- **Learn how to use**
  - Templates
  - Namespaces
  - Collections from the Standard Library



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

What Is the Standard Library?

- Template Refresher
- Namespaces
- Sequence Containers
- Associative Containers
- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Standard Library

- **C++ defines its *Standard Library* (SL)**
  - Input/output, streams, and collections
  - Extended in C++11 with threads and synchronization, regular expressions, reference counters, etc.
- **The SL raises the level of how you write C++**
  - Provides powerful, efficient mechanisms for functionality that previously had to be handcrafted



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

What Are Collections?

- **Collections are a set of specifications for class templates for manipulating collections of objects**
- **The specification dictates performance requirements, as well as signatures**
  - The sort algorithm must evaluate the ordering predicate `X<Y` at most `n*log(n)` times, where `n` is the number of objects
- **The SL is designed to be *minimal***
  - A class is not defined if it can be implemented in terms of others
  - *Adapters* are used to create new classes from basic ones
- **The main components of the collections implementation of the SL are *containers*, *iterators*, and *algorithms***

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?

Template Refresher

- Namespaces
- Sequence Containers
- Associative Containers
- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Templates

- **Templates are used to create a family of functions or classes**
  - Differing only in the type of data they act on
- **Function templates are used to replace preprocessor `#define` macros commonly used in C**
  - The preprocessor is a very naïve text-substitution mechanism that induces great risks of unintended mistakes
- **Class templates are most frequently used to implement collections**
  - More often used to implement a library, such as SL, than by application programmers

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Function Templates

- **Function templates are written as functions parameterized for one or more types**
- **It is not necessary to instantiate a function template explicitly**
  - The compiler generates the instantiation from a call
  - Programmer may explicitly instantiate if required

```
template <class T>      // or <typename T>
T max(T left, T right) {
  return left > right ? left : right;
}
```

```
int a, b;
float x, y;

cout << max(a, b);  // OK
cout << max(x, y);  // OK
cout << max(a, y);  // will not compile
cout << max<int>(a, y);  // OK
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Class Templates

- **Class templates must be explicitly instantiated**

```
template <class T>
class Array {
 public:
  Array(int s):size(s), elements(new T[s]){}

  T& operator[](int i){
    return elements[i];
  }

 private:
  T* elements;
  int size;
};
```

```
Array<float> myFloatArray(5);
myFloatArray[4] = 42.0;

typedef Array<int> IntArray;
IntArray myIntArray(3);
myIntArray[0] = 0;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Which Types Can be Used in a Template?

- **It all depends on how it is used in the template**
  - In `max` below, the type `T` has to have a `>` operator
  - If it hasn't, the compiler will flag an error
  - Unfortunately, the error will be reported in the body of `max`, not at the place where `max` is called
  - This leads to very hard-to-understand error messages, particularly in more complex templates written by somebody else; e.g., the standard library

```
template <class T>
T max(T left, T right) {
  return left > right ? left : right;
}
```

```
Thing t1, t2, t3; // no > operator for Thing
//…
t1 = max(t2,t3);
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

C++20 *Concept*

- **A *concept* is a template that can define requirements**

```
template <typename T>
concept GreaterComparable = requires (T x) { x > x; };
```

```
GreaterComparable max(GreaterComparable left, GreaterComparable right) {
  return left > right ? left : right;
}
```

```
Thing t1, t2, t3; // no > operator for Thing
//…
t1 = max(t2,t3); // error is flagged here since Thing is not GreaterComparable
```

- The error is now flagged at the point where `max` is called incorrectly

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher

Namespaces

- Sequence Containers
- Associative Containers
- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Namespaces

- **The Standard Library uses the *namespace* feature**
  - All SL identifiers are in **`namespace`** `std`
- **A namespace provides a scope for names**
  - Used to avoid name clashes
- **For example, the Standard Library and a proprietary library might both declare the class "string"**
  - No problem if each appears within its own namespace
- **A namespace definition looks like this**

```
namespace myScope{
  ...
}
```

Anything that has a name can be declared in a namespace; e.g., classes, functions, variables, or nested namespaces

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Namespaces

- **A namespace can be added to but cannot contain the same name more than once**

Legal...

```
namespace myScope{
  int x;
  class C1{
  };
}
...
namespace myScope{
  int y;
  class C2{
  };
}
...
```

Illegal...

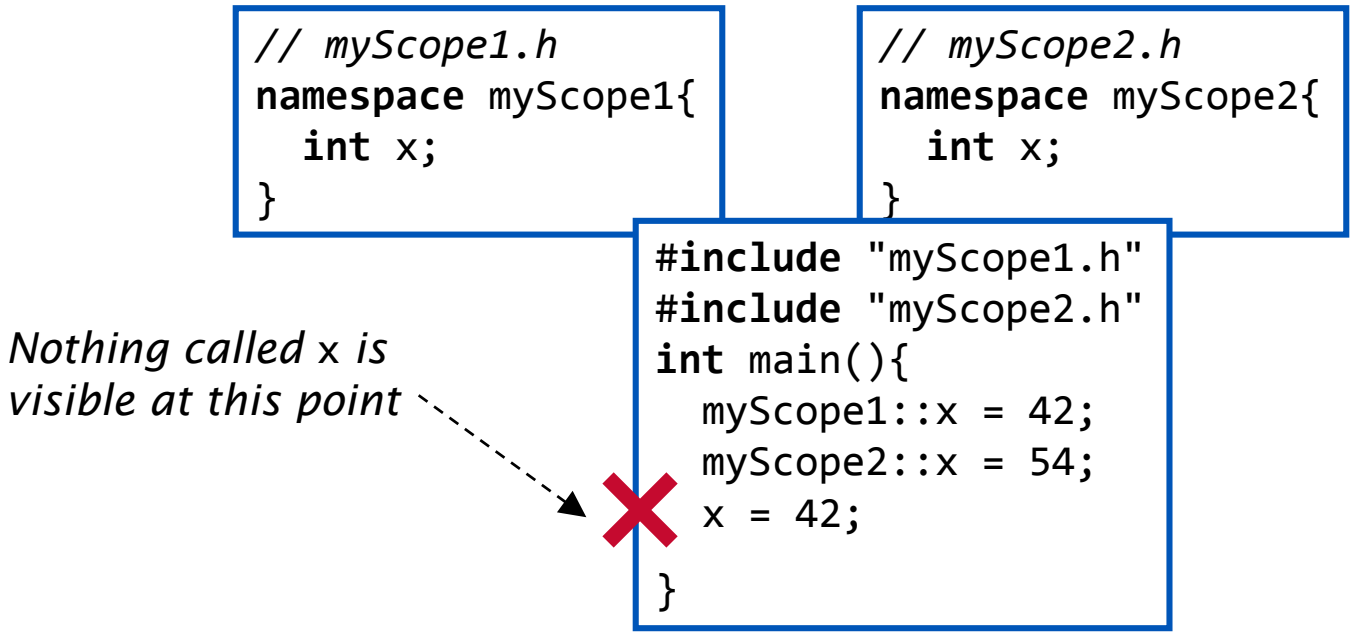
```
namespace myScope{
  int x;
  class C1{
  };
}
...
namespace myScope{
  int x;


}
...
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Accessing Namespace Members

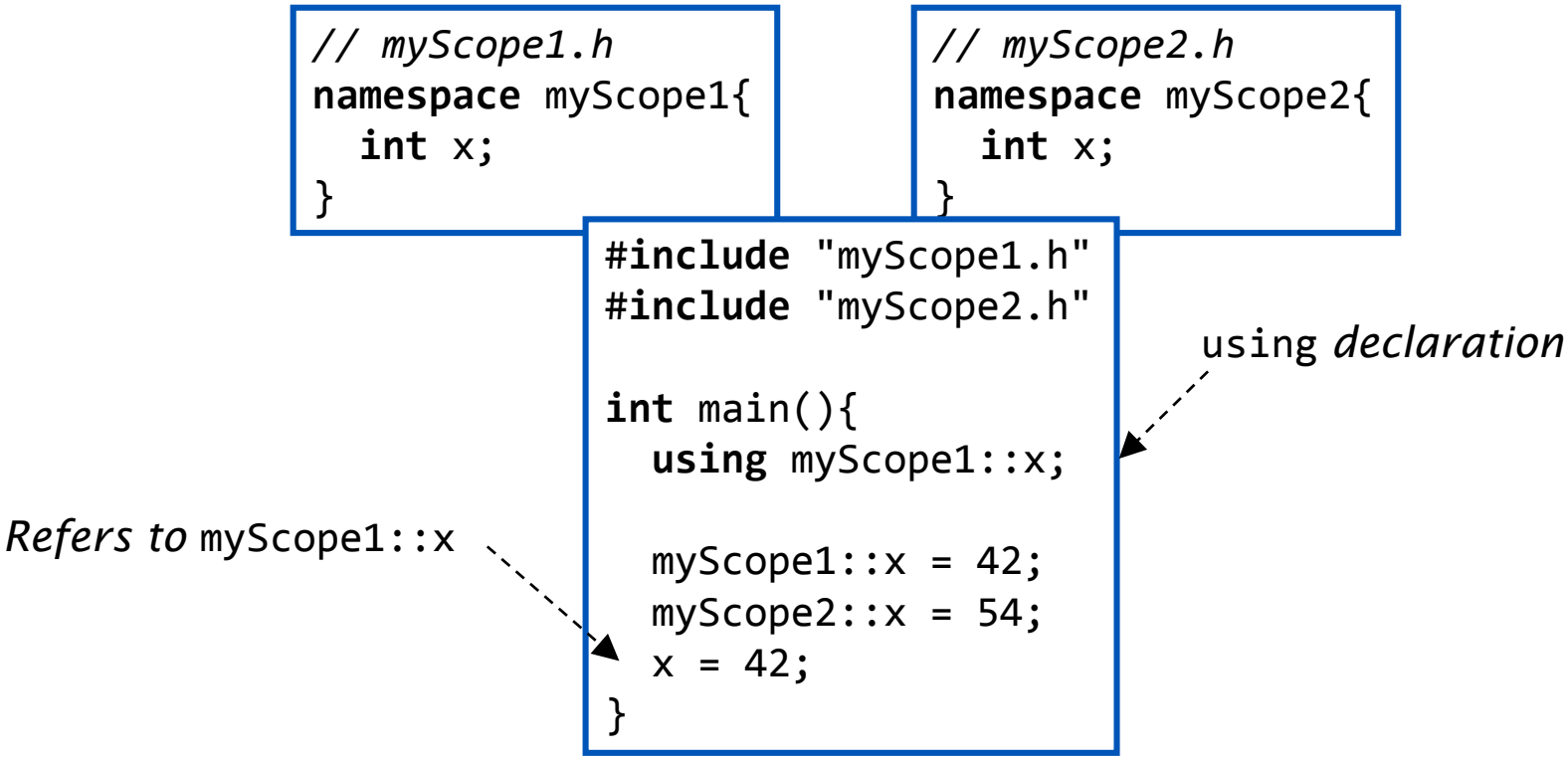
- **A namespace member can be accessed using the scope operator**
  - As if the namespace name were a class name
  - Names not declared in a namespace exist in the global namespace
- **Different namespaces can contain the same name**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Making Namespace Members Visible

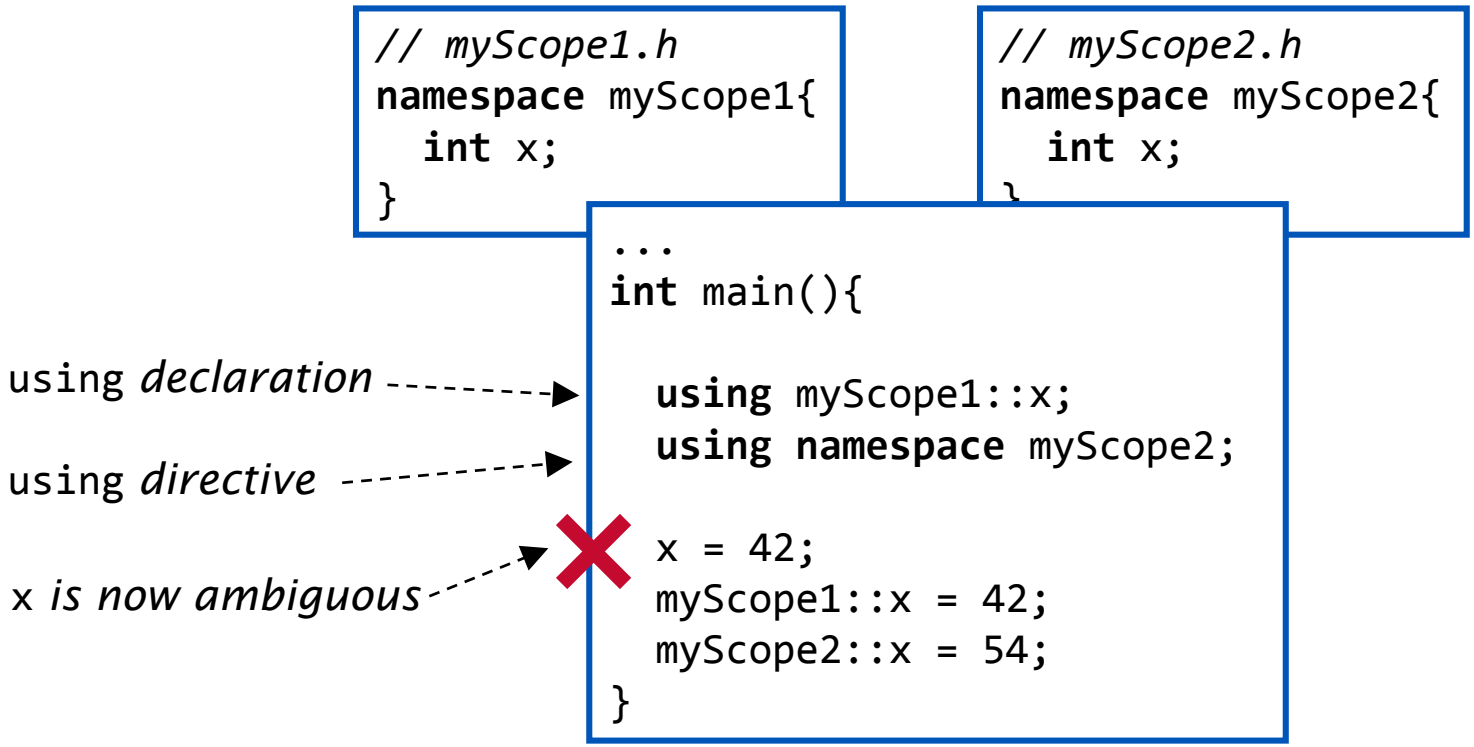
- **A `using` declaration promotes a single namespace member to the *current namespace*; i.e., makes it visible to the compiler**
- **Note that `using` obeys the same scoping rules as declarations**
  - Avoid file scope where possible



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Making Namespace Members Visible

- **A `using` directive promotes all members of a namespace to the current namespace**
  - This can introduce name clashes



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Unnamed Namespaces

- **A `namespace` can be defined without a name**
  - Unnamed namespaces appearing in different compilation units are considered different
    - This is intended to replace the use of `static` as a means of declaring file-scope variables
  - There is no way to access a member of an unnamed namespace from another compilation unit

- **What does this program print?**

```
// main.cpp
#include <iostream>
...
#include "f.h"
int main(){
  f();
  cout << x;
}
```

```
// f.h
namespace {
  int x = 0;
}

void f();
```

```
// f.cpp
#include "f.h"

void f(){
  x = 1;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces

Sequence Containers

Introduction

- Iterators
- Algorithms
- Function Objects and Lambdas
- Adapters

- Associative Containers
- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Containers

- **A *container* is an object that contains other objects**
  - SL containers are templates that can be instantiated for different classes or types
  - Containers typically provide methods for inserting, removing, and finding objects, with the same names for all containers
  - Not all containers implement every method
- **There are two categories of containers**
  - *Sequence* containers
    - An ordered collection of objects
    - SL provides `vector`, `list`, and `deque`
    - New in C++11: `forward_list`, `array`
  - *Associative* containers
    - A collection that can be accessed using a *key*
    - SL provides `set`, `map`, `multiset`, and `multimap`
    - New in C++11: unordered (hashtable) versions of the above

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sequence Containers

- **A `vector` of `Apples` is declared as shown below**
  - All SL declarations are in the scope of **`namespace`** `std`

```
#include <vector>

std::vector<Apple> myApples;
```

or...

```
#include <vector>
using std::vector;

vector<Apple> myApples;
```

or...

```
#include <vector>
using namespace std;

vector<Apple> myApples;
```

- **`deque` and `list` are declared in a similar manner**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Characteristics of Sequence Containers

- **A `vector` behaves like a traditional array, except its size may be changed**
  - Fast, random access
- **New in C++11: `array`**
  - Fixed size and efficient, but STL-compatible
- **A `deque` is a double-ended queue**
  - Fast insertion and deletion at either end
  - Random access is allowed, but is slower than `vector`
- **A `list` provides fast insertion and deletion in the middle, as well as at either end**
  - No random access
  - Doubly linked list
- **New in C++11: `forward_list` is singly linked**
  - More efficient than `list`, only forward links
- **As far as possible and appropriate, the interfaces are the same**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sequence Container Operations

- **The sequence containers provide operations to insert, access, and remove elements**
  - `T` is the class for which the template is being instantiated
- **As far as reasonable, all sequence containers offer the same methods, e.g.:**

```
void push_back(const T& x);
void pop_back();
T& back();
const T& back() const;
T& front();
const T& front() const;
```

- **`list`, `deque`, and `forward_list` also provide:**

```
void push_front(const T& x)
void pop_front();
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sequence Container Operations

- **`vector`, `array`, and `deque` provide:**

```
T& at(size_t n);
const T& at(size_t n) const;
T& operator[](size_t i);
const T& operator[](size_t i) const;
```

- **Note that `at()` is required to throw an exception of type `out_of_range` if the vector is accessed out of bounds, but `operator[]()` is not**
- **`vector` also provides:**

```
void resize(size_t n);
```

- **All containers (except `forward_list`) provide `size_type size() const;`**
  - Returns the number of elements in the container
- **Why doesn't `operator[]()` throw an exception when the index is out of range?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sequence Container Operations: Example

```
#include <vector>
using namespace std;

vector<Apple> myApples;

int main(){
  Apple a1, a2, a3, a4;

  myApples.push_back(a1);
  myApples.push_back(a2);
  myApples.push_back(a3);

  myApples.resize(4);

  myApples[3] = a4;

  ...
}
```

- What value is in `myApples[3]` after the `resize` call?

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

C++11 Initializer Lists

- **In C++11, initializer lists were added, so even the library containers may be initialized with many values at the same time, like arrays**
  - At the same time, a uniform initialization syntax was added using braces **{}**
  - This new syntax can be used anywhere
  - It provides stricter checks for initialization values than the old syntax

```
vector<int> vi = {4,6,7,8,2,11,0};
vector<int> vj{4,6,7,8,2,11,0};   //  = not needed

vector<int> vk{};                 // default constructor

int i{};                          // zero initialized
short s{1000000};                 // error, value too large
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces

Sequence Containers

- Introduction

Iterators

- Algorithms
- Function Objects and Lambdas
- Adapters

- Associative Containers
- More About Iterators and Algorithms



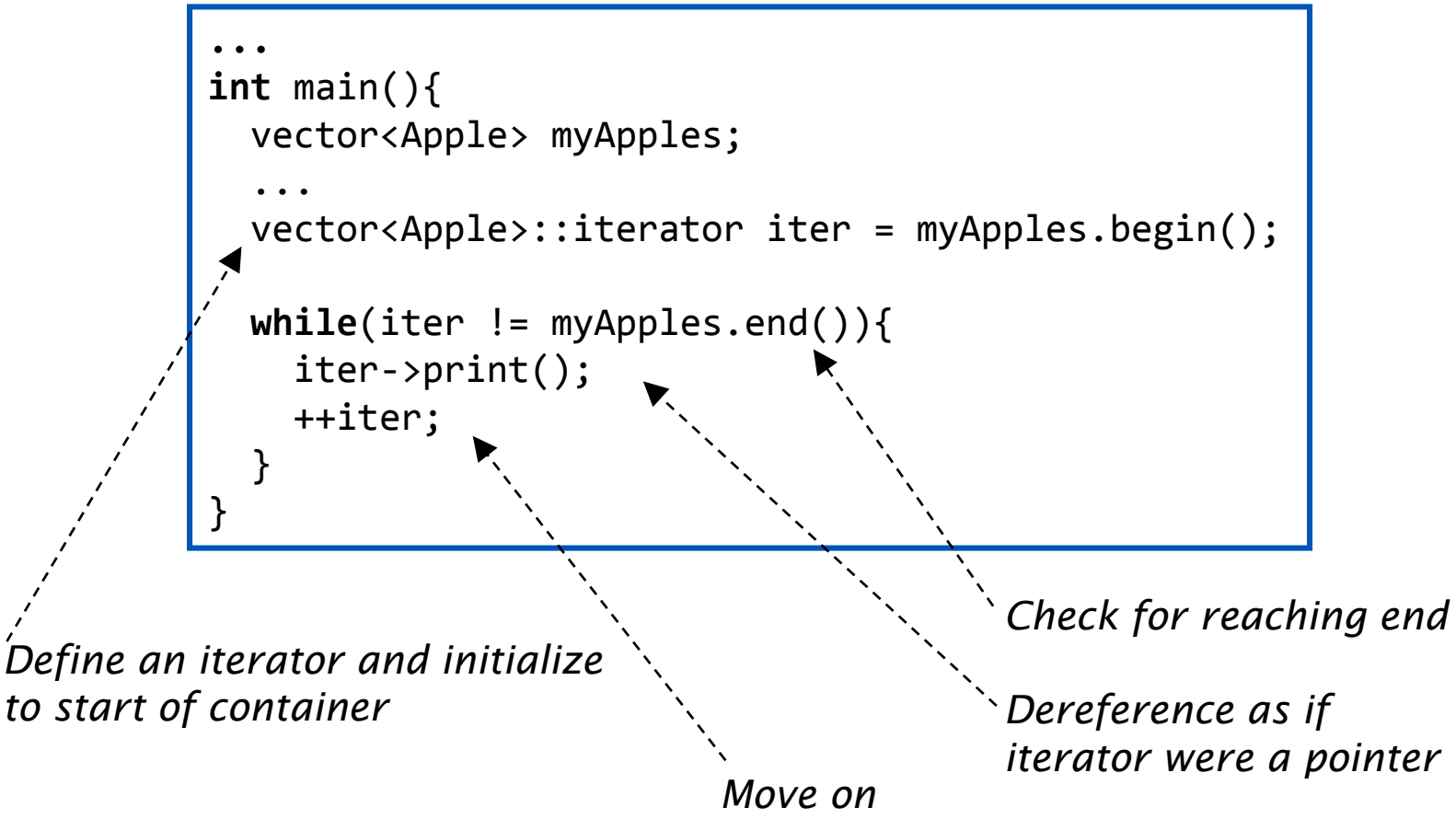
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Iterators

- **An *iterator* is an object used to traverse a container**
  - Iterators are designed to be used like pointers to array elements
- **Various types of iterators permit different operations; e.g.:**
  - Movement in forward and reverse directions
  - Access to the current element of the container
  - Stepping `n` steps
- **Containers have member functions that return iterators; e.g.:**
  - `begin()` returns an iterator pointing to the first element
  - `end()` returns an iterator pointing *beyond* the last element
    - The iterator returned by `end()` does *not* point to an element in the container
- **Iterators may become invalid when a container is modified**
  - Do not create an iterator until you are ready to use it
- **Why does `end()` point one element *beyond* the last one?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Iterators



- **What does `vector<Apple>::iterator` mean?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: auto Declarations

- **In C++11, the compiler can figure out the type of a declared variable**
  - It must be initialized
  - The type is inferred by the compiler
  - The variable is *not* dynamically typed; it cannot change its type later

```
auto i = 34;                // not much gained here, i becomes an int



auto iter = myApples.begin(); // from previous page

                            // lots of typing saved here
                            // also, if myApples changes type,
                            // iter will follow
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: The Range `for` Statement

- **Iterating over entire collections is common and rather verbose**
- **C++11 supports "for each" statements like many other languages**
  - Iterate from "beginning to end," although you can **`break`**
  - All SL containers can be used, as can an array, and anything for which you have a `begin()` and `end()`; e.g., an `istream`

```
vector<Apple>  myApples;
...
for(const auto & a : myApples)
{
  a.print();
}
```

Use `&` *without* **`const`** *to modify the* `Apples`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces

Sequence Containers

- Introduction
- Iterators

Algorithms

- Function Objects and Lambdas
- Adapters

- Associative Containers
- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Algorithms

- ***Algorithms* are functions defined by the SL that can be applied to elements of a container**
  - The elements are specified by an *iterator-range*
  - The algorithms thus do not know the container type they are working on
  - No specialized version of the algorithm for each container!
- **The main categories of algorithms are**
  - Sorting operations; e.g., `sort`
  - Non-modifying sequence operations; e.g., `find`, `for_each`
  - Modifying sequence operations; e.g., `swap`
  - Numeric operations; e.g., `accumulate`
- **SL algorithms are declared in the header `<algorithm>`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The sort Algorithm

- The simplest form of the sort algorithm uses operator<, which must be defined for the objects being sorted

```
class Apple{
public:
  bool operator<(const Apple&) const;
  ...
};
```

```
#include "apple.h"

#include <vector>
#include <algorithm>
...
vector<Apple> myApples;
...
sort(myApples.begin(), myApples.end());
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The sort Algorithm

- **To take more control of the sort operation, provide a compare function as parameter**
  - To sort a collection of pointers according to the objects in the collection, not according to the pointer values
  - The third argument to sort is a function pointer

```
class Apple{
public:
  bool operator<(const Apple&) const;
  ...
};

bool compareApplePtrs(
          const Apple* l,
          const Apple* r)
{
  return (*l) < (*r);
}
```

```
...
vector<Apple*> myApplePtrs;
...
sort(myApplePtrs.begin(),
     myApplePtrs.end(),
     compareApplePtrs);
```

- **What would happen if we omitted the third argument to sort?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The for_each Algorithm

- **This algorithm evaluates a function or function object for each element of the collection**
  - Allows iteration over all elements without the use of a loop

```
class Apple{
public:
  void print() const;
  ...
};
```

```
void applePtrPrinter(const Apple* pa){
  pa->print();
}
```

```
vector<Apple*> myApplePtrs;
...
for_each(myApplePtrs.begin(),
         myApplePtrs.end(),
         applePtrPrinter);
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Iterator Categories

- **The container functions `begin()` and `end()` will return iterators of different categories**
  - `vector` yields *random access iterators*
  - `forward_list` yields *forward iterators*
- **`sort` requires random access iterators**
  - A `forward_list` thus cannot be sorted with algorithm `sort`
    - Signaling that this would be a very inefficient way of sorting the list
- **The iterator categories form a hierarchy of operation capabilities**
  - The random access iterator is the most capable, and can thus be used with all algorithms

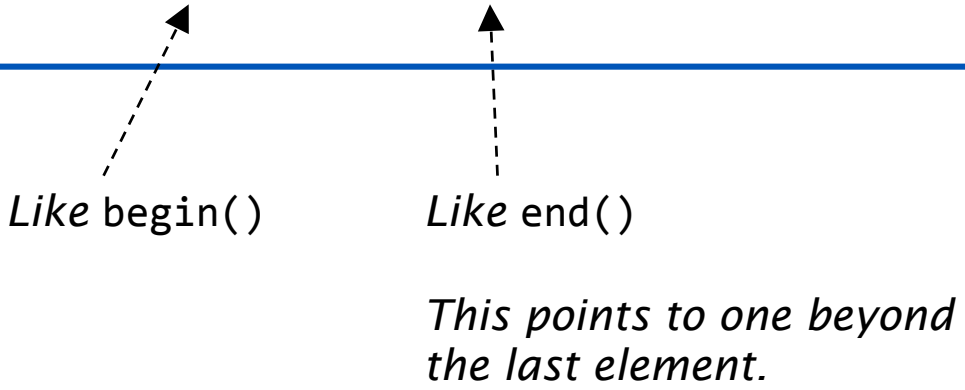
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Iterators and Pointers

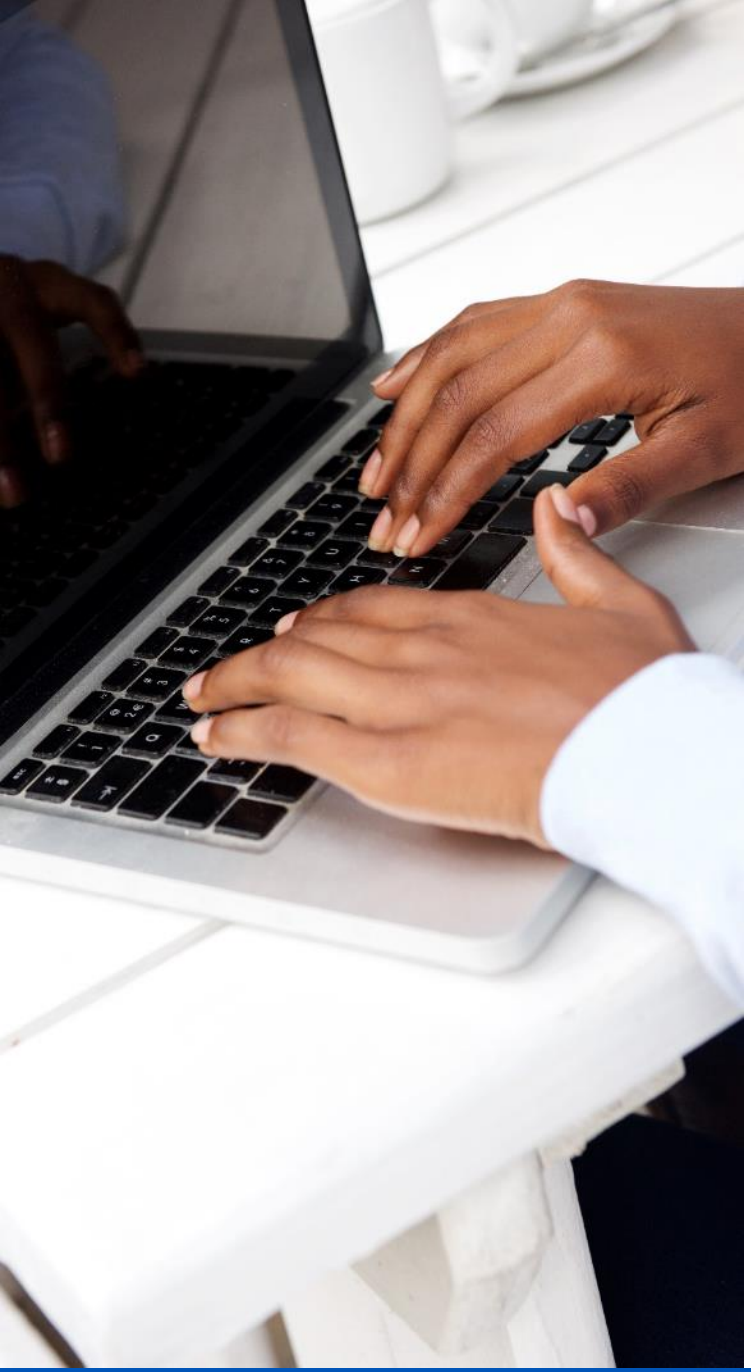
- **Iterators have been modeled after pointers**
  - Iterators are pointer-like; for example, they have `operator++`, `operator--`, `operator*`
- **That means pointers are iterator-like!**
  - Algorithms will accept pointers as if they were iterators

```
int numbers[100];
...

sort(numbers, &numbers[100]);
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 3.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 3.1: Containers, Iterators, and Algorithms

In this exercise, you will be working on a console application

You will

- Populate a `vector` of objects
- Print the contents in forward and reverse orders
- Sort the contents
- Explore STL in any way you like

All of this can be done using the STL, without necessarily needing to write loops

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces

Sequence Containers

- Introduction
- Iterators
- Algorithms

Function Objects and Lambdas

- Adapters

- Associative Containers
- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Function Objects

- **Many algorithms will accept either the address of a global function or a *function object* (also knows as *functor*) as a parameter**
  - For example, the `sort` and `for_each` algorithms
- **A function object is more powerful than a function pointer**
  - A function object can remember its own state
  - And can do anything else an object can do
- **The class of the function object defines the method `operator()`**
  - That makes objects callable, as if they were functions
  - Since it is a class object, it can also use all other class mechanisms
    - Other methods and data
    - Constructors
    - Inheritance

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Function Objects

```
class Apple{
public:
  bool operator<(const Apple&) const;
  ...
};

class ApplePtrComparator{
public:
  bool operator()(const Apple* l, const Apple* r) {
    ++count;
    return (*l) < (*r);
  }
...
};
```

```
vector<Apple*> myApplePtrs;
...
ApplePtrComparator cmp;
sort(myApplePtrs.begin(),
     myApplePtrs.end(),
     cmp);
```

Send the function object to `sort`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Standard Function Objects

- **To sort a collection of apples in reverse order, we could write an appropriate function object**
- **However, this is such a common requirement, it is provided by the SL**
  - Use the library class template `greater<T>`
  - Assumes an `operator>` for the class `T`
- **The header `<functional>` provides this and many others**

We create an anonymous temporary object of class `greater<Apple>`

```
#include <algorithm>
#include <functional>
...
vector<Apple> myApples;
...
sort(myApples.begin(),
     myApples.end(),
     greater<Apple>() );
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: Lambda Expressions

- **A *lambda expression* is best described as an "in-line function object"**
  - Provides a way of specifying a function object directly as a parameter to a function; e.g., `sort`
  - A lambda expression always starts with `[]`, then the parameter list, and ends with the `{}`-bracketed code
  - The advantage is that the comparison code is located directly where it will be used, at the expense of learning yet another language feature and its syntax

```
#include <algorithm>
...
vector<Apple> myApples;
...
sort(myApples.begin(), myApples.end(),
     [](const Apple& l, const Apple& r){return r < l;}
);
                  // sorts the Apples in descending order
```

Why not `l > r` *here ?*

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces

Sequence Containers

- Introduction
- Iterators
- Algorithms
- Function Objects and Lambdas

Adapters

- Associative Containers
- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Container Adapters

- **A *container adapter* is a template that provides an interface for a commonly used data structure**
  - The implementation is based on a separate container class; default is `deque`
- **For example, the `stack` adapter takes two parameters**
  - The class of objects to be stacked
  - The class of container the stack will adapt to its use

```
// file <stack> (part of SL)
template<class T, class Cont = deque<T> >
class stack{
  ...
};
```

```
#include <stack>
...
stack<Apple> myAppleDequeStack;
stack<Apple, list<Apple>> myAppleListStack;
stack<Apple, vector<Apple>> myAppleVectorStack;
```

Pre-C++11 code needs a space here, so it is not confused with the operator >>

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Container Adapters

- **The container being adapted does not have to be part of the SL**
  - Any class providing the necessary members is acceptable
  - `vector`, `list`, and `deque` are all acceptable to `stack`
  - You can also write your own
    - Example:

```
template<class T, class Cont = deque<T> >
class stack{
  public:
    void push(const T& obj){c.push_back(obj);}
    ...
  private:
    Cont c;
};
```

```
template <class T>
class MyContainer {
public:
  MyContainer(){...}
  void push_back(const T& obj){...}
  ...
};
```

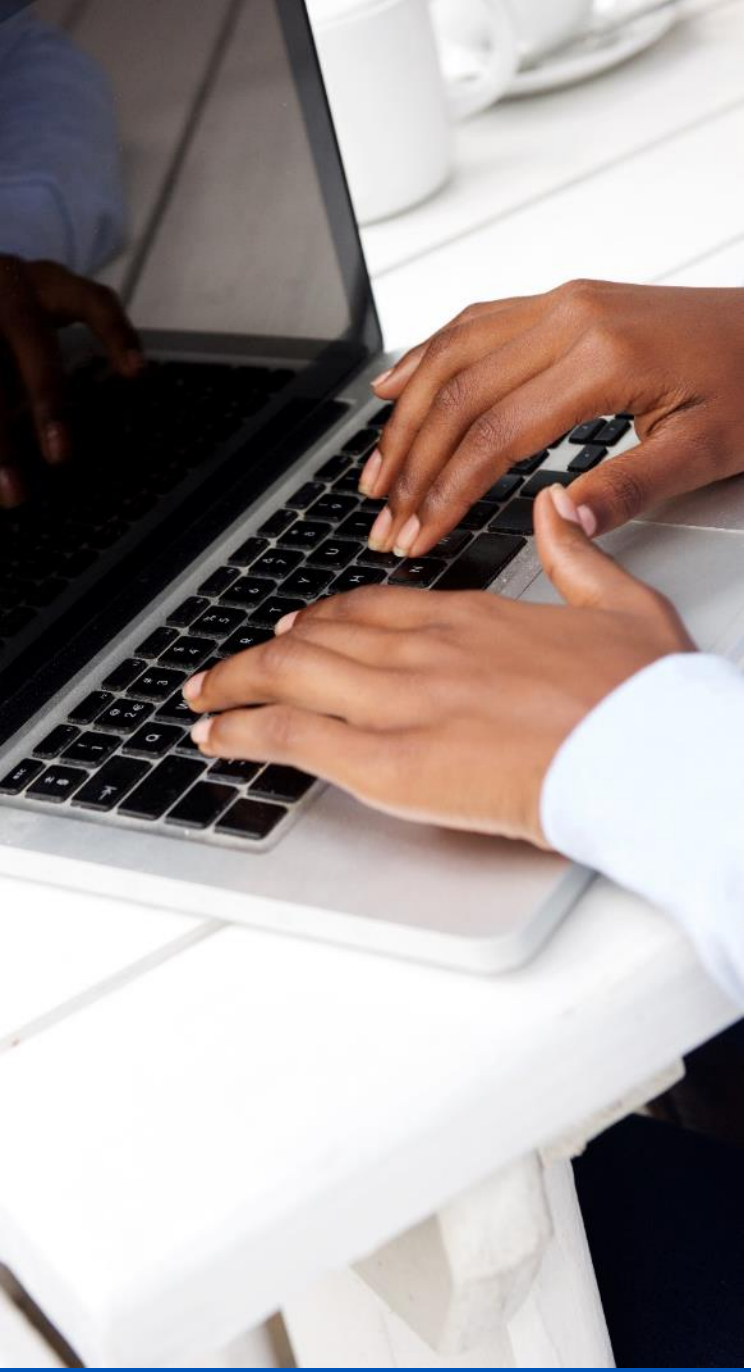
```
stack<Apple, MyContainer<Apple>> myStack;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Other Adapters

- **The container adapters provided by the SL are**
  - `stack`
  - `queue`
  - `priority_queue`
- **There are also *iterator adapters***
  - `reverse_iterator` reverses the direction of iteration
  - `front_insert_iterator` and `back_insert_iterator` make space available at the front or back of a container

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 3.2

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 3.2: Function Objects

In this exercise, you will extend the program you wrote for Exercise 3.1, using function objects and lambdas

You will

- Use function objects and lambda expressions to allow sorting of a collection using a variety of keys

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces
- Sequence Containers

Associative Containers

Introduction

- `set` and `multiset`
- `map` and `multimap`

- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Associative Containers

- **An *associative container* allows an element to be addressed using a key value**
  - Rather than an index as used by a sequence container
  - The container stores the objects in its own order
- **Like a sequence container, an associative container can also be traversed sequentially using an iterator**
- **The SL provides four types of associative containers**

	No duplicate keys allowed	Duplicate keys allowed	Declared in header
Element acts as its own key	`set`	`multiset`	`<set>`
Key is separate from element	`map`	`multimap`	`<map>`

- **C++11 has new, faster, unordered versions of these**
  - `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Section Example

- **On the following slides, we will use associative containers to manipulate collections of objects of the following class:**

```
class Y{
public:
  Y(string n=""):name(n){
  }
  bool operator<(const Y& rhs) const{
    return name < rhs.name;
  }
  void print() const{
    std::cout << name;
  }
  ...
private:
  string name;
  ...
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sort Order

- **The elements of an ordered associative container are sorted on insertion**
- **By default, `operator<` is used to determine sort order**
  - Alternatively, a function object may be supplied
- **The function that determines sort order is also used to determine whether objects are duplicates**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sequential Read Access Using Iterators

- **Associative containers can be read sequentially using iterators**
  - Like sequence containers

```
set<Y> myYSet;  // or multiset
...
for(auto iter = myYSet.begin();
    iter != myYSet.end();
    ++iter){
  iter->print();
}
```

- **Cannot use iterators to modify associative containers**
  - Object position is determined by sort order, not order of insertion
    - Use `insert` function

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces
- Sequence Containers

Associative Containers

- Introduction

`set` and `multiset`

- `map` and `multimap`

- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Declaring the Container

- **A `set` or `multiset` is declared as follows:**

```
set<Y> myYSet;
multiset<Y> myYMultiSet;
```

  - `Y::operator<` will be used to determine sort order and duplication check

- **Alternatively, a *function object class* may be supplied**

```
class YCompare{
public:
  bool operator()(const Y& l, const Y& r)const{
    return ...; // return true if l considered less than r
  }
};
```

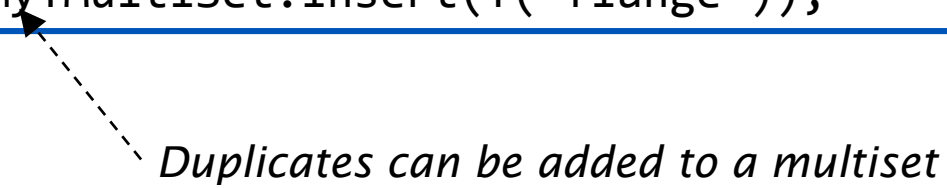
```
set<Y, YCompare> myYSet;
multiset<Y, YCompare> myYMultiSet;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inserting an Object Into a `multiset`

- **Member function `insert(const Y& arg)` places a copy of `arg` into the container**
  - Returns an `iterator` that points to the inserted object
    - Usually ignored

```
multiset<Y> myYMultiSet;
myYMultiSet.insert(Y("sprocket"));
myYMultiSet.insert(Y("flange"));
myYMultiSet.insert(Y("flange"));
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inserting an Object Into a `set`

- **Member function `set::insert(const Y& arg)` places a copy of `arg` into the container**
  - The insertion will fail if the argument is a duplicate of an object already in the set
- **`set::insert` returns `pair<iterator, bool>`**
  - `pair` is a template defined by the SL to be used whenever a function needs to return two values
  - This returned *pair* contains an iterator pointing to the element and an indication of whether the insertion succeeded
- **A `pair` object simply contains two public data members, called `first` and `second`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inserting an Object Into a set

- Check the second member of the returned pair to see if inserting into a set was successful

```
typedef set<Y> SetY;
typedef pair<SetY::iterator, bool> ResultType;
SetY myYSet;
...
ResultType r = myYSet.insert(Y("X234"));
   // auto r would be even simpler
if(r.second){
  cout << "Insert of " << r.first->print() << " succeeded\n";
}
else
{
  cout << "Insert failed\n";
}
```

- Under what circumstances would this insertion fail?

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Finding an Object

- **`find(const Y& arg)` returns an `iterator` pointing to the first element that matches the argument**
  - In a `set`, there can be only one such element
  - If no element is found, `end()` is returned

```
SetY myYSet;  // or multiset<Y>
...

Y match("flange");

auto iter = myYSet.find(match);

if(iter != myYSet.end()){
  cout << "found matching item" << endl;
  iter->print();
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

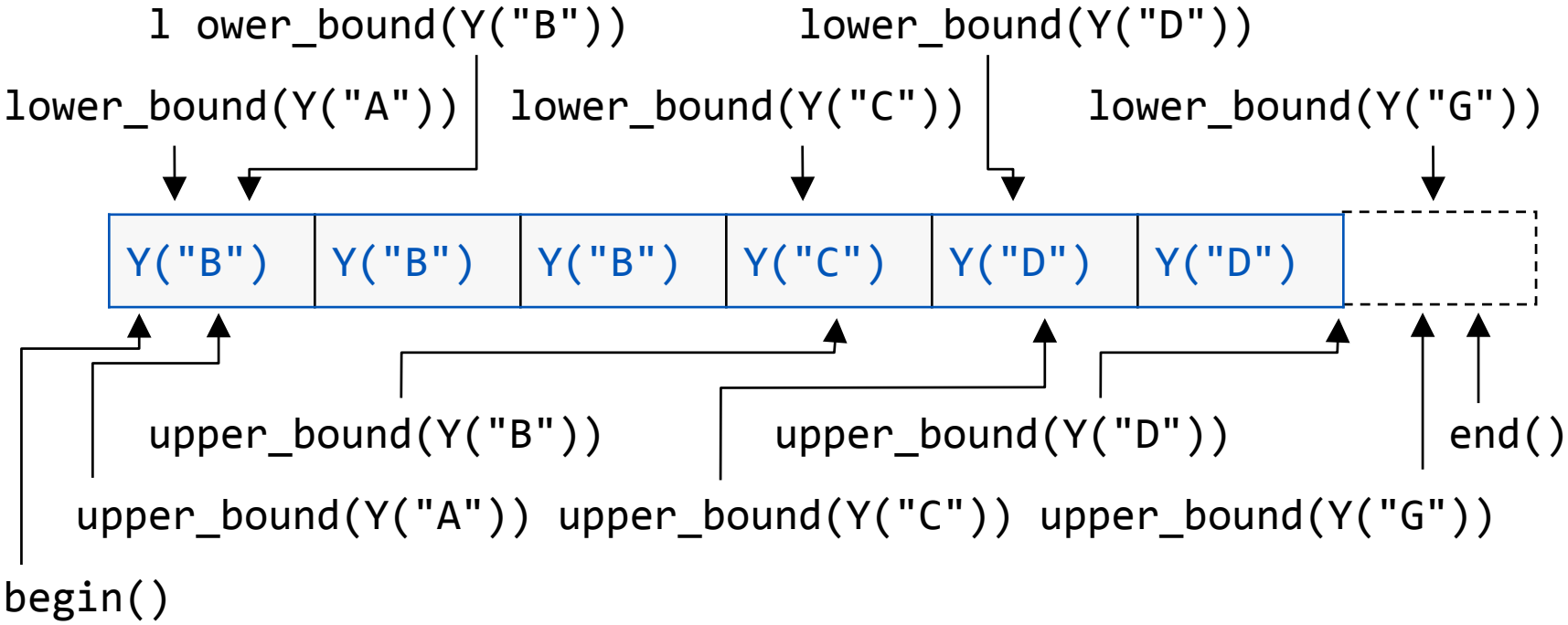
Finding a Range of Objects

- **`lower_bound (const Y& arg)` returns an `iterator`**
  - Points to the first element greater than or equal to `arg`
  - Returns `end()` if no suitable element is found
- **`upper_bound (const Y& arg)` returns an `iterator`**
  - Points to just past the last element less than or equal to `arg`
  - Returns `begin()` if no suitable element is found
- **`equal_range (const Y& arg)` returns a `pair<iterator, iterator>`**
  - Equivalent to calling `lower_bound` and `upper_bound` for the same `arg`
  - Returns `begin()` and/or `end()` where appropriate

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Finding a Range of Objects

- **These functions are used together to access all elements with a given key or range of keys**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Finding a Range of Objects

- For example, to find all `Y` between `"B"` and `"D"`

```
multiset<Y> myYSet;
...
for(auto iter = myYSet.lower_bound(Y("B"));
    iter != myYSet.upper_bound(Y("D"));
    ++iter)
{
  iter->print();
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces
- Sequence Containers

Associative Containers

  - Introduction
  - `set` and `multiset`
  - `map` and `multimap`

- More About Iterators and Algorithms



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Declaring a `map` or `multimap`

- **The `map` and `multimap` templates take two types as arguments**
  - First argument type is the *`key_type`*
  - Second argument type is the *`referent_type`*
  - For example, to associate a collection of `Y` with `string` keys:

```
map<string, Y> mySYMap;
multimap<string, Y> mySYMultiMap;
```

- **Again, a *function object class* may be supplied**

```
map<string, Y, MyStringCompare> mySYMap;
multimap<string, Y, MyStringCompare> mySYMultiMap;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inserting an Object Into a `multimap`

- **`map` and `multimap` templates include the following declaration:**

```
typedef pair<const key_type, referent_type> value_type;
```

- **Key-referent pairs are created using the global template function `make_pair` and placed in the container using the member function `insert`**
  - `make_pair` is a function template that will make a `pair` of any two values

```
typedef multimap<string, Y> MultiMapSY;
MultiMapSY mySYMultiMap;

mySYMultiMap.insert(make_pair("PART03", Y("flange")));
mySYMultiMap.insert(make_pair("DEV06", Y("sprocket")));
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inserting an Object Into a `map`

- **Member function `map::insert(const value_type& arg)` places a copy of `arg` into the container**
  - The insertion will fail if the argument has a key that is already in the `map`
- **`map::insert` returns `pair<iterator, bool>`**
  - Returns an iterator pointing to the element and an indication of whether the insertion succeeded

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inserting an Object Into a `map`

- **Check the `second` member of the returned `pair` to see if inserting into a `map` was successful**

```
map<string, Y> mySYMap;
...
auto result = mySYMap.insert(make_pair("ID04", Y("X2")));
if(result.second){
  cout  << "Insert of "
        << result.first->second.print()
        << " succeeded\n";
}
else {
  cout << "Insert failed\n";
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Finding Objects in a `map` or `multimap`

- **`find`, `lower_bound`, `upper_bound`, and `equal_range` can be used in the same way as in a `set` or `multiset`, except that**
  - The parameter is of *`key_type`*
  - The returned `iterator` refers to a `value_type` pair

```
map<string, Y> myMap;

...

auto iter = myMap.find("ID06");

if(iter != myMap.end()){
  cout << "found matching item" << endl;
  iter->second.print();
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

map::operator[]

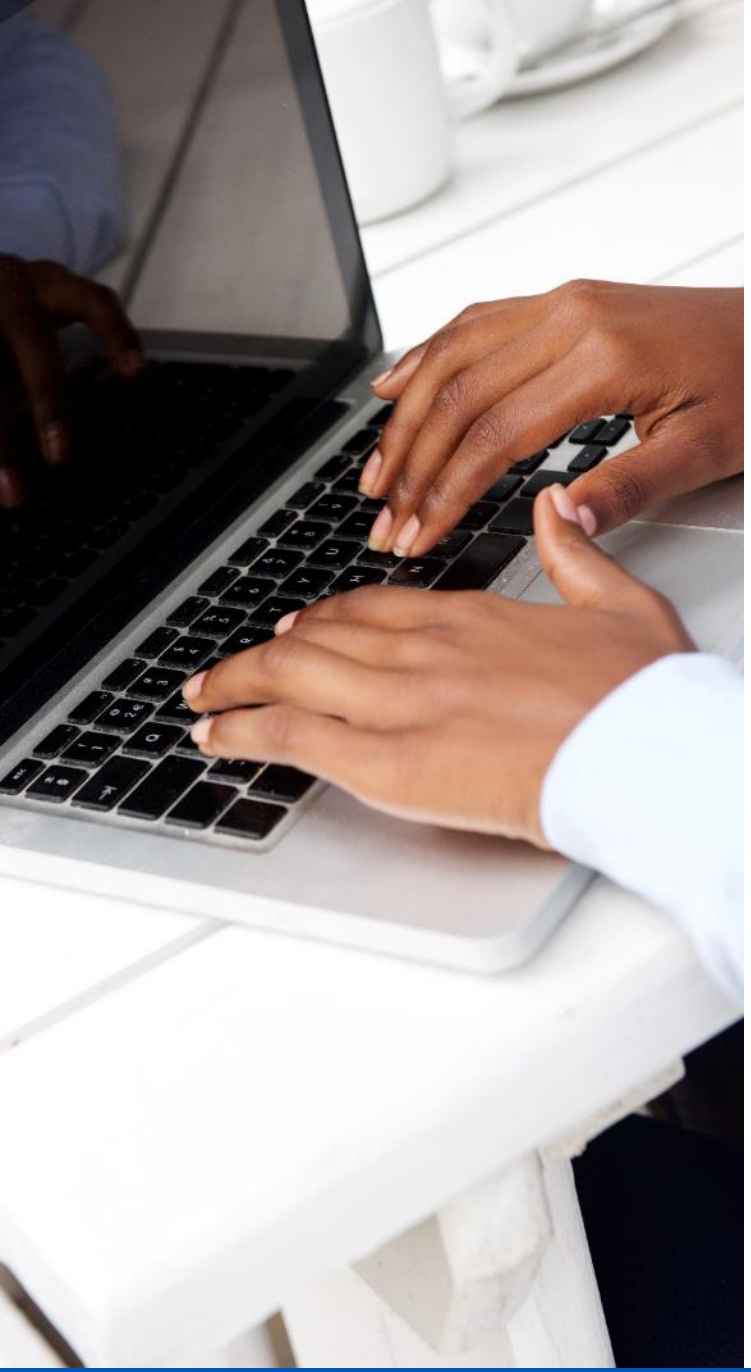
- **map provides operator[](const *key_type&*)**
  - Returns a reference to the *referent_type*
- **This allows a map to be used with an array-like idiom**
  - A key can be treated as if it were an index
- **Warning: If an object with the given key is not found, one is created using the default constructor**
  - A subscript operator lookup will compile only if the referent type has a default constructor

```
map<string, Y> myMap;

myMap.insert(make_pair("ID06", Y("wheel")));
myMap["DEV07"] = Y("sprocket");
Y yVar = myMap["ID06"];
myMap["WHAT?"].print();
```

- **What will happen at the `print()` call?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 3.3

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 3.3: Associative Containers

In this exercise, you will extend the program you wrote for Exercise 3.2 using associative containers

You will

- Use `set` and `multiset` templates to store and retrieve objects using the objects themselves as keys
- Use `map` and `multimap` templates to store and retrieve objects using separate keys

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces
- Sequence Containers
- Associative Containers

More About Iterators and Algorithms

Insert Iterators

- Catalog of Algorithms and Function Objects



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Writing Through an Iterator

- **Writing to a container through a normal iterator overwrites the existing element**

```
...

int main(){
  list<int> myList;

  myList.push_back(3);

  *myList.begin() = 5;

  // print contents of myList
  ...
}
```

- **What would this program print?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Insert Iterators

- **The SL provides three types of *insert iterators* and three function templates to make them easy to use**
  - Provided through the header `<iterator>`

```
int main(){
  list<int> myList;

  myList.push_back(3);
  back_inserter(myList) = 4;
  front_inserter(myList) = 2;
  inserter(myList, myList.begin()) = 1;
  inserter(myList, myList.end()) = 5;

  // print contents of myList
  ...
}
```

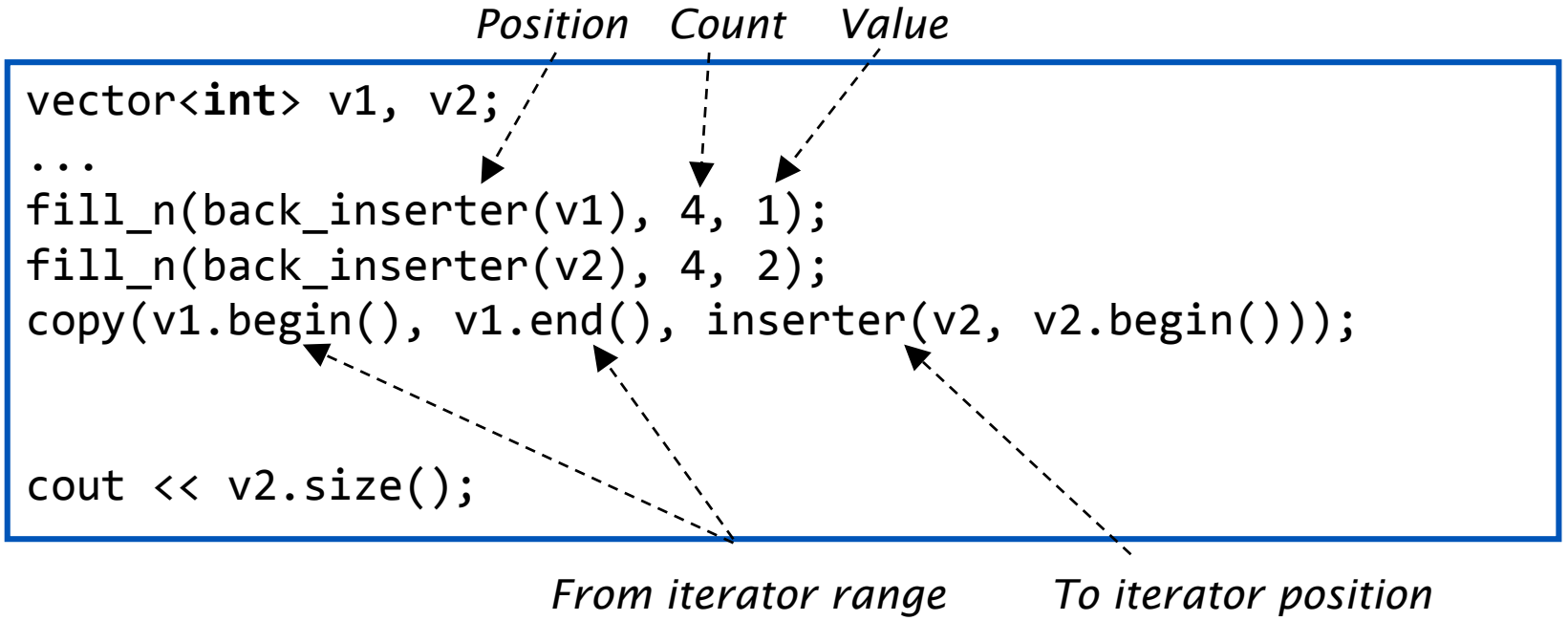
Uses `push_front`, *so couldn't be used on a vector*

Inserts element in front of the location given by second argument; can be used on a vector

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Using Insert Iterators

- **Insert iterators are primarily used with algorithms that allow operations to be performed on multiple elements; for example:**
  - `fill_n` fills elements of a container with a value
  - `copy` copies values from an iterator range to an iterator position



- **What do v1 and v2 now contain?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- What Is the Standard Library?
- Template Refresher
- Namespaces
- Sequence Containers
- Associative Containers

More About Iterators and Algorithms

- Insert Iterators

Catalog of Algorithms and Function Objects



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Non-Modifying Sequence Operations Provided in <algorithm>

`for_each()`	Do operation for each element of a sequence
`find()`	Find first occurrence of a value in a sequence
`find_if()`	Find first match of a predicate in a sequence
`find_first_of()`	Find a value from one sequence in another
`adjacent_find()`	Find an adjacent pair of values
`count()`	Count occurrences of a value in a sequence
`count_if()`	Count matches of a predicate in a sequence
`mismatch()`	Find the first elements for which two sequences differ
`equal()`	True if the elements of two sequences are pairwise equal
`search()`	Find the first occurrence of a sequence as a subsequence
`find_end()`	Find the last occurrence of a sequence as a subsequence
`search_n()`	Find *n* consecutive occurrences of a value in a sequence

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Modifying Sequence Operations Provided in `<algorithm>`

`transform()`	Apply an operation to every element in a sequence
`copy()`	Copy a sequence starting with its first element
`copy_backward()`	Copy a sequence starting with its last element
`swap()`	Swap two elements
`iter_swap()`	Swap two elements pointed to by iterators
`swap_ranges()`	Swap elements of two sequences
`replace()`	Replace elements with a given value
`replace_if()`	Replace elements matching a predicate
`replace_copy()`	Copy sequence replacing elements with a given value
`replace_copy_if()`	Copy sequence replacing elements matching a predicate
`fill()`	Replace every element with a given value
`fill_n()`	Replace first *n* elements with a given value

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Modifying Sequence Operations Provided in `<algorithm>`

`generate()`	Replace every element with the result of an operation
`generate_n()`	Replace first *n* elements with the result of an operation
`remove()`	Remove elements with a given value
`remove_if()`	Remove elements matching a predicate
`remove_copy()`	Copy a sequence removing elements with a given value
`remove_copy_if()`	Copy a sequence removing elements matching a predicate
`unique()`	Remove equal adjacent elements
`unique_copy()`	Copy a sequence removing equal adjacent elements
`reverse()`	Reverse the order of elements
`reverse_copy()`	Copy a sequence into reverse order
`rotate()`	Rotate elements
`rotate_copy()`	Copy a sequence into a rotated sequence
`random_shuffle()`	Move elements into a uniform distribution

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sorted Sequence Operations Provided in `<algorithm>`

`sort()`	Sort with good average efficiency
`stable_sort()`	Sort, maintaining relative order of equal elements
`partial_sort()`	Get the first part of sequence into order
`partial_sort_copy()`	Copy getting the first part of output into order
`nth_element()`	Put the *n*th element in its proper place
`lower_bound()`	Find the first occurrence of a value
`upper_bound()`	Find the last occurrence of a value
`equal_range()`	Find a sub-sequence with a given value
`binary_search()`	Is a given value in a sorted sequence?
`merge()`	Merge two sorted sequences
`inplace_merge()`	Merge two consecutive sorted sub-sequences
`partition()`	Place elements matching a predicate first
`stable_partition()`	Place elements matching a predicate first, preserving relative order

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Set and Heap Operations Provided in <algorithm>

- **Set operations**

`includes()`	True if a sequence is a sub-sequence of another
`set_union()`	Construct a sorted union
`set_intersection()`	Construct a sorted intersection
`set_difference()`	Construct a sorted sequence of elements in the first but not the second sequence
`set_symmetric_difference()`	Construct a sorted sequence of elements in one but not both sequences

- **Heap operations**
  - These operations keep a sequence in a state that allows fast sorting

`make_heap()`	Make sequence ready to be used as a heap
`push_heap()`	Add element to heap
`pop_heap()`	Remove element from heap
`sort_heap()`	Sort the heap

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Selection and Permutation Operations Provided in `<algorithm>`

- **Selection operations**

`min()`	Smaller of two values
`max()`	Larger of two values
`min_element()`	Smallest value in sequence
`max_element()`	Largest value in sequence

- **Permutation operations**

`next_permutation()`	Next permutation in lexicographical order
`prev_permutation()`	Previous permutation in lexicographical order

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Numeric Operations Provided in `<numeric>`

`accumulate()`	Accumulate results of operation on a sequence
`inner_product()`	Accumulate results of operation on two sequences
`partial_sum()`	Generate sequence by operation on two sequences
`adjacent_difference()`	Generate sequence by operation on a sequence

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Predicates Provided in <functional>

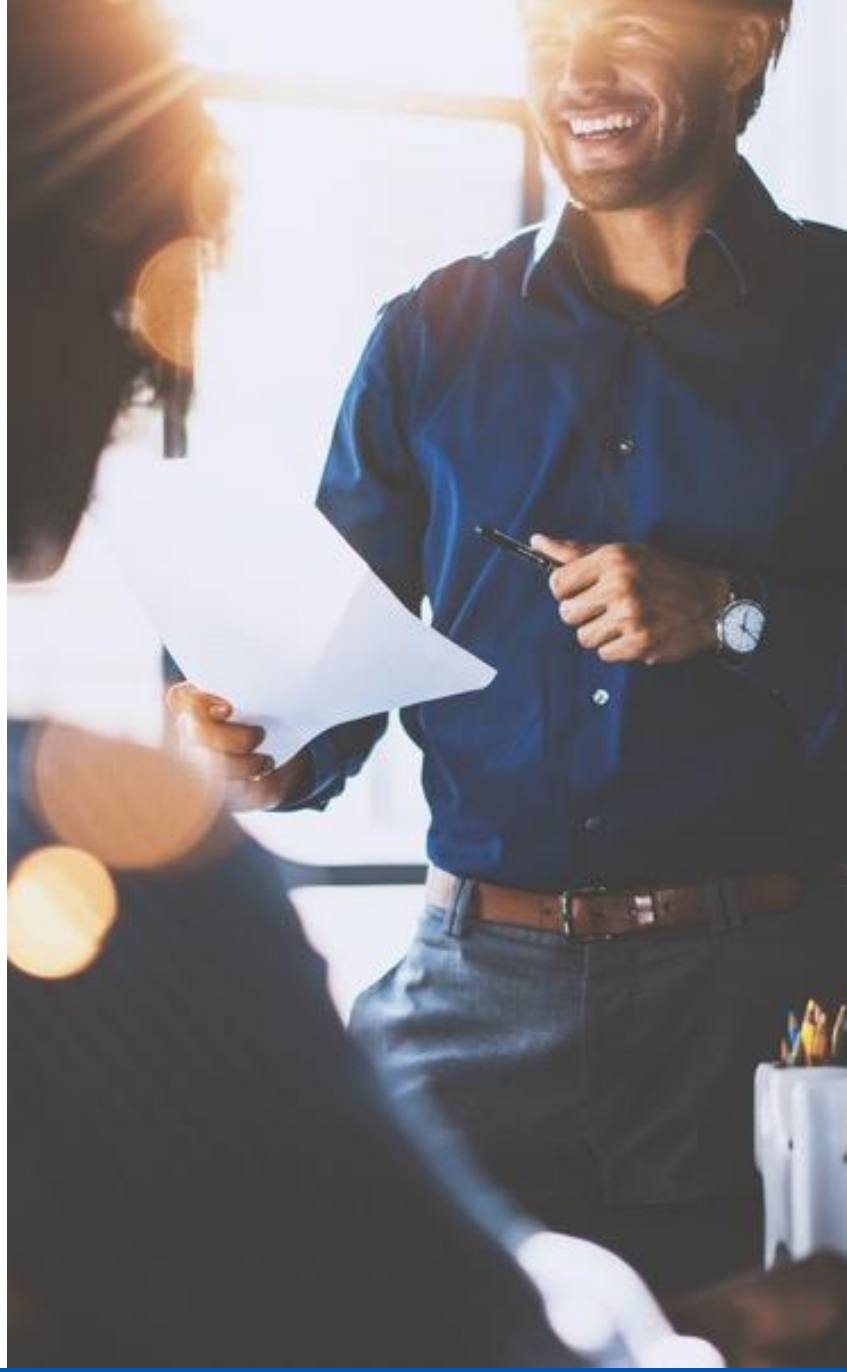
- **A predicate is a function object for which `operator()` returns `bool`**
  - Middle column indicates whether the function takes one or two parameters

`equal_to`	Binary	`arg1==arg2`
`not_equal_to`	Binary	`arg1!=arg2`
`greater`	Binary	`arg1>arg2`
`less`	Binary	`arg1<arg2`
`greater_equal`	Binary	`arg1>=arg2`
`less_equal`	Binary	`arg1<=arg2`
`logical_and`	Binary	`arg1&&arg2`
`logical_or`	Binary	`arg1||arg2`
`logical_not`	Unary	`!arg`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- **Learn how to use**
  - Templates
  - Namespaces
  - Collections from the Standard Library



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



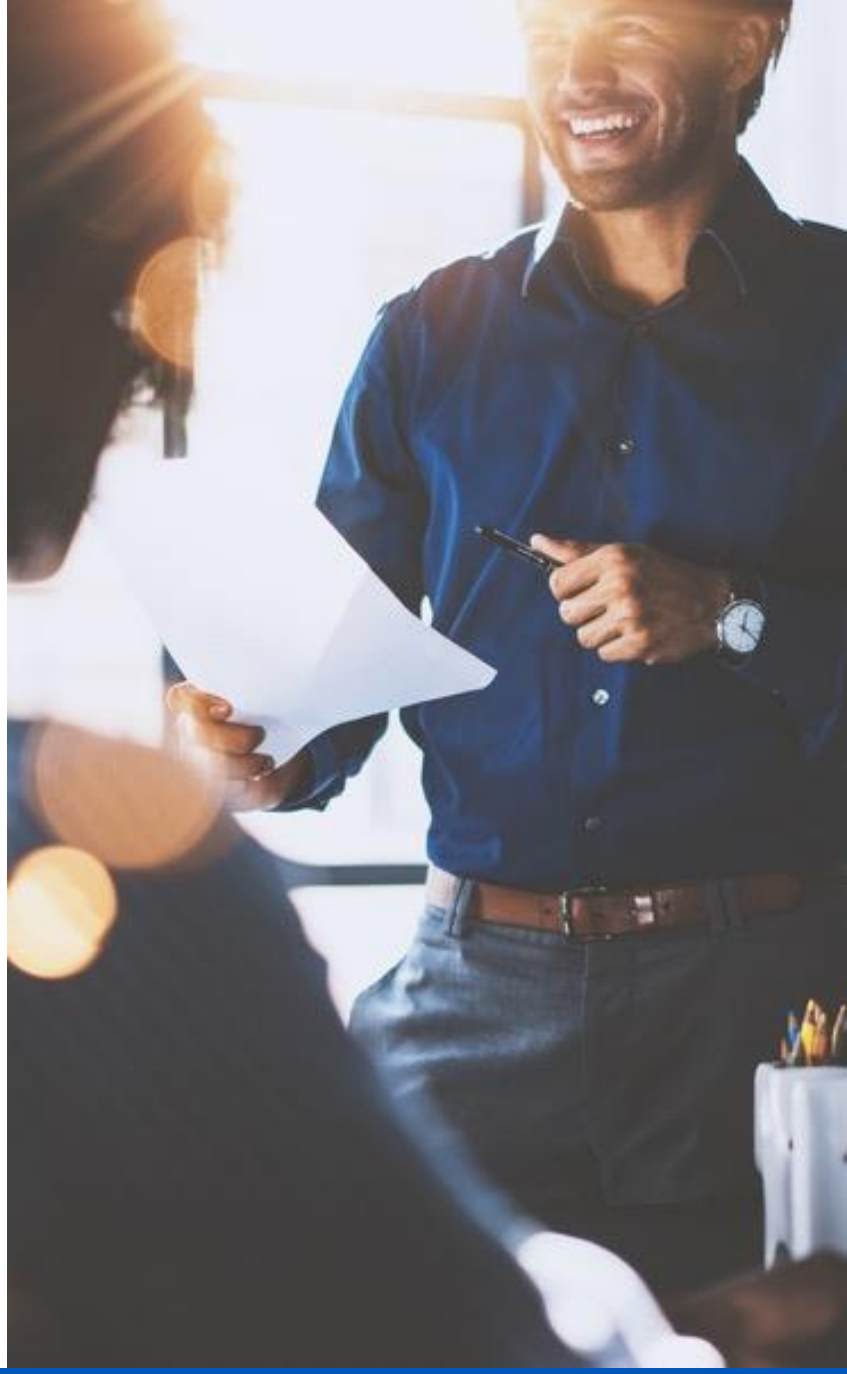
Chapter 4

Patterns, Part 2: Observer Pattern

LEARNING TREE INTERNATIONAL

Objectives

- **Learn how the Observer pattern works**
  - Why it is prevalent in modern GUI system designs
- **Learn how C++ features are used in variations of the Observer pattern**
  - Exceptions
  - Runtime type information
  - Casts



GUI = graphical user interface

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Observer Pattern

Identification

- Problem
- Solution
- Consequences

- **Exceptions**
- **Runtime Type Information**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Identification

- **Name**
  - Observer
- **Classification**
  - Behavioral
  - Object
- **Intent**
  - Define a one-to-many dependency between objects
  - When one object changes state, all of its dependents are notified and updated automatically
- **Also known as**
  - Publish/subscribe
  - Model/view
  - Document/view

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Observer Pattern

- Identification

Problem

- Solution
- Consequences

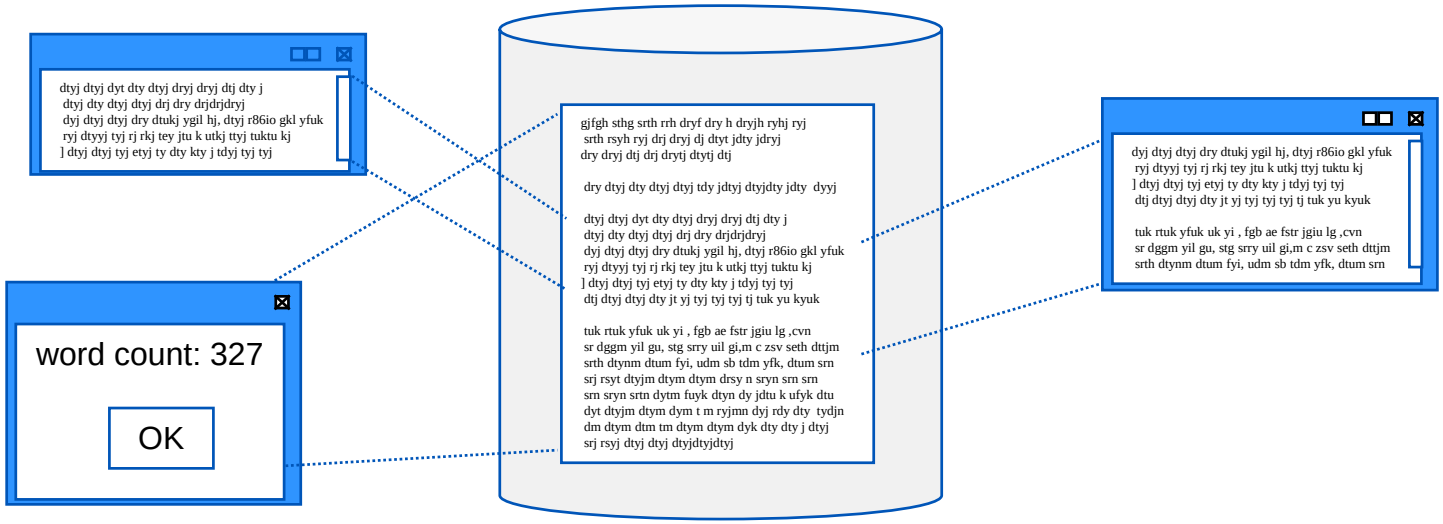
- **Exceptions**
- **Runtime Type Information**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Problem

- **Motivation**
  - Many systems have many views of the same data; e.g.:
    - Multiple windows displaying the same document
  - Coupling the objects too closely reduces reusability
    - OK for the window object to know about the document object
    - Document object should not know about the window object
    - Yet needs to inform the windows when it changes
    - If document needs to know about windows, it will be hard to add new kinds of windows



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Problem

- **Applicability**
  - When an abstraction has two aspects and one is dependent on the other
    - A document and a view of that document
  - A change to one object requires that an indeterminate number of other objects be notified
  - An object needs to notify other objects without making assumptions about their class
- **Known uses**
  - Smalltalk classes `Model` and `View`
  - Java classes `Observer` and `Observable`
  - Apple calls them `Publisher` and `Subscriber`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Observer Pattern

- Identification
- Problem

Solution

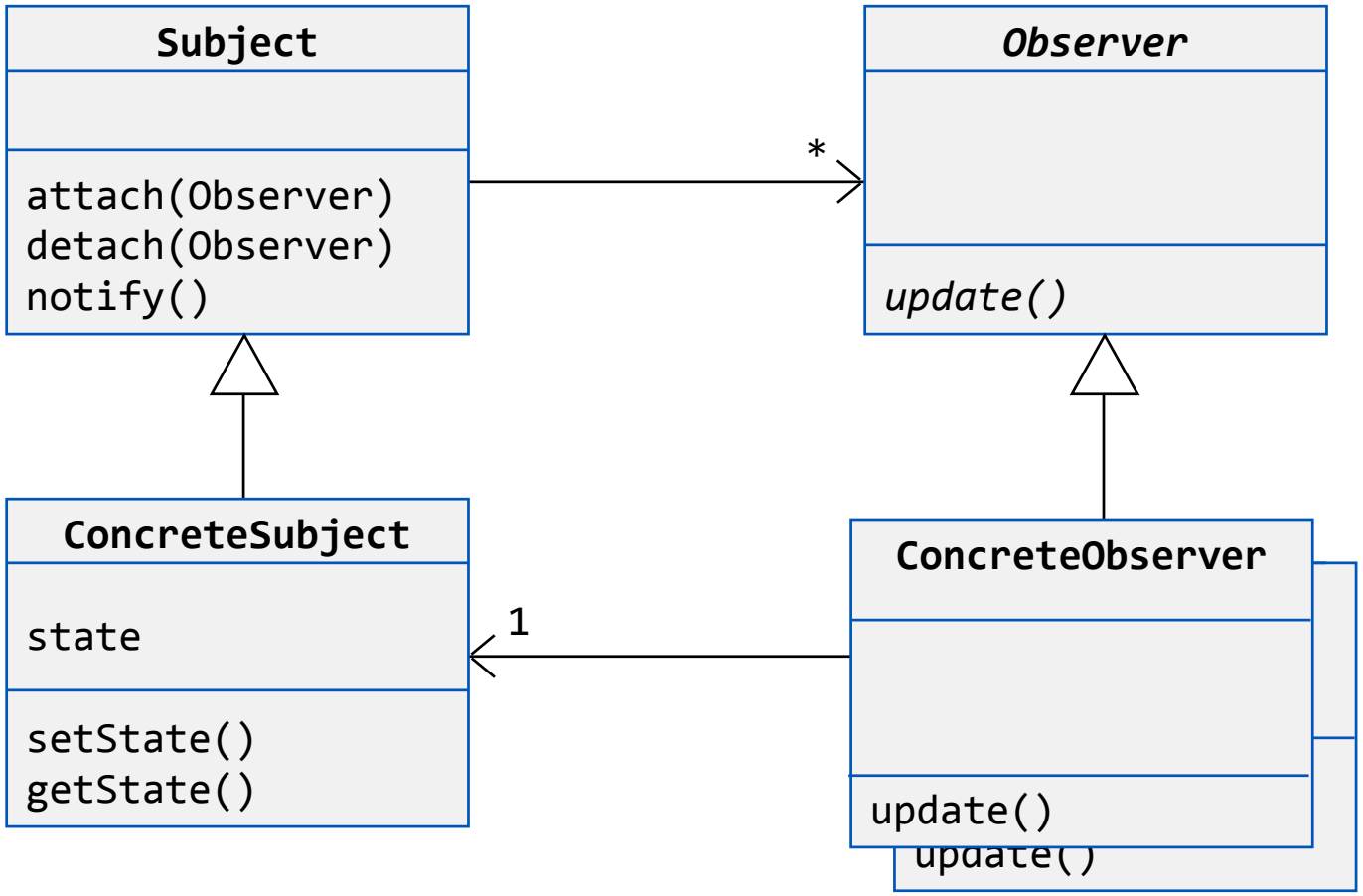
- Consequences

- **Exceptions**
- **Runtime Type Information**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Structure



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Participants

- **Subject**
  - Maintains a collection of pointers to its observers
  - Provides an interface for attaching, detaching, and notifying observers
- **Observer**
  - Declares an interface for being notified when the subject changes
- **Concrete subject**
  - Provides an implementation for modifying and accessing its state
  - Notifies its observers when it changes
- **Concrete observer**
  - Polymorphically defines the observer notification implementation

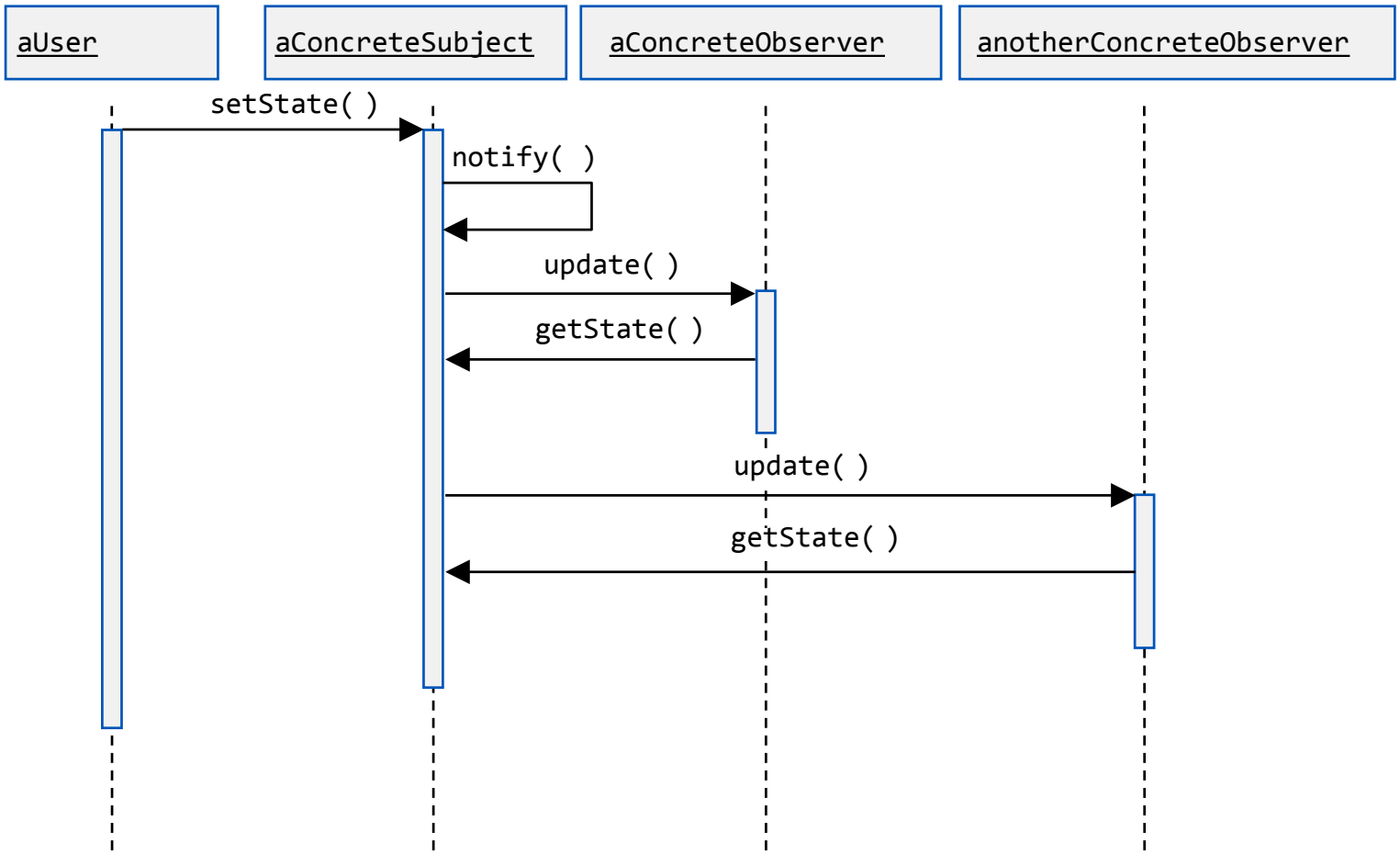
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Collaborations

- **Concrete observers attach to and detach from subjects as required**
- **When a concrete subject's state is changed, it notifies all of its observers**
  - Done via code in the `Subject` class
- **The concrete observer requests the state of the concrete subject and updates itself accordingly**
  - Variations of this pattern allow state information to be passed to the observer in the notification

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Collaborations



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Observer Pattern

- Identification
- Problem
- Solution

Consequences

- **Exceptions**
- **Runtime Type Information**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code

- **These classes can form part of a general-purpose library**
  - The code is not dependent on the application that uses it

```
class Observer;

class Subject {
public:
  void attach(Observer &);
protected:
  void notify() const;
private:
  std::set<Observer*> myObs;
};
```

```
class Observer{
public:
  virtual void update() = 0;
};
```

```
void Subject::notify() const {
  for(auto obs: myObs){
    obs->update();
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code

- **These classes are part of a specific application**

```
class MySubject: public Subject{
public:
  ...
  void toggleState(){
    state = !state;
    notify();
  }

  bool getState() const{
    return state;
  }

private:
  bool state;
};
```

```
class MyObserver: public Observer{
public:
  MyObserver(MySubject* s): sub(s){
    sub->attach(*this);
  }

  void update(){
    bool curState = sub->getState();
    // act accordingly
  }

private:
  MySubject* sub;
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Consequences

- **Advantages**
  - Abstract coupling between subject and observer
  - A subject can communicate with observers without knowing the exact class of the observer
  - Communication is in the reverse direction to visibility
    - The concrete subject can talk to its concrete observers without knowing anything about them
- **Disadvantages**
  - Observer may receive notification for changes it is not interested in
  - In the simplest form of this pattern, the observer will not know what has changed
    - Must maintain its own reference to its subject
  - If no new observer will ever be added, the pattern just adds complexity
  - Variations of this pattern can be used to minimize these problems

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Hands-On Exercise 4.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 4.1: Observer Pattern

- **In this exercise, you will work on the pelican crossing application**
  - A real pelican crossing has poles on each side of the road that hopefully each display the same light states!
  - This is a perfect opportunity to use the Observer pattern: Decouple the control logic from the `Poles` showing the lights
- **You will**
  - Write the classes `Subject` and `Observer`
  - Derive class `Pole` from class `Observer` and implement `Pole::update()`
  - Derive class `Pelican` from class `Subject` and call `Subject::notify()` from `Pelican`
  - Modify the program to create two or more `Poles`, attached to a single `Pelican`
  - Create a new kind of `Observer` and see how easy it is due to the reduced dependencies

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- **Observer Pattern**

Exceptions

Introduction

- Syntax
- Control Flow
- Effective Exception Handling

- **Runtime Type Information**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exceptions

- **Exceptions provide a means of communicating errors or other conditions**
  - Cannot be bypassed or forgotten by mistake
  - Allow handling to take place at any desired level
  - Separate error-handling code from normal processing
- **Exceptions play a particularly important role in the production of robust libraries**
  - The user cannot be relied on to check error returns or flags
- **Where is the best place to handle an exception?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

`throw`, `try`, and `catch`

Exception syntax is based on three keywords:

- **`throw`**
  - Used to raise an exception at the location an error is discovered
  - An exception is always raised synchronously—it will never be thrown out of the blue
- **`try`**
  - Marks the beginning of a block of code where we want to catch an exception
- **`catch`**
  - Identifies a block of code used to handle an exception

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- **Observer Pattern**
- **Exceptions**
  - Introduction
  - **Syntax**
    - Control Flow
    - Effective Exception Handling
- **Runtime Type Information**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The `throw` Statement

- **A `throw` statement raises an exception**
  - Syntax is: `throw` *`expression`*`;`
  - *`expression`* can evaluate to a value of either a built-in or class type
- **Statements following the `throw` are not executed**
  - Control passes to the end of the enclosing `try` block

```
double arcsine(double sine){
  if(sine < -1.0 || sine > 1.0){
    throw string("arcsine: out of range");
  }
  // calculate and return angle
  ...
}
```

Not executed if exception is raised

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The try Block

- Code that may cause an exception to be raised should be enclosed in a try block




```
void func(double x){
  try{
     double angle = arcsine(x);
     cout << "Angle = " << angle << endl;
  }
  catch(string& str){
     cerr << str << endl;
  }
  cout<< "What did you see?" << endl;
}
```

- A try block is followed by one or more catch clauses

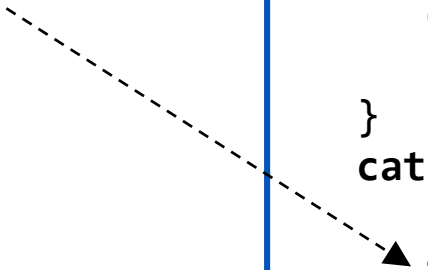
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The catch Clause

- A catch clause is declared as either
  - `catch(`*type name*`) {`*statements*`}`

  or
  - `catch(...) {`*statements*`} // catches anything`

Rethrows the same exception



```
void func(double x){
   try{
      double angle = arcsine(x);
      cerr << "Angle = " << angle << endl;
   }
   catch(string& str){
      cerr << str << endl;
      throw;
   }
   cerr << "What did you see?" << endl;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The `catch` Clause

- **When an exception is raised, `catch` clauses are tested in the order they are declared**
- **The first *matching* clause is executed**
  - Control then passes to the end of all of the clauses
  - If no match is found, the exception is *propagated*
- **A `catch` clause matches an exception if**
  - Its parameter is an *exact* match for the exception type
    - **`throw`** `42;` will not be caught by **`catch`**`(float f){}`
  - Its parameter is an accessible base class of the exception type
  - It is declared as **`catch`**`(...){`*`statements`*`}`, in which case it matches any exception
    - This must come as the last **`catch`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple catch Clauses

```
// base class for all library exceptions
class exception {
  ...
};
```

```
void func(double x){
  try{
    double angle = arcsine(x);
    cerr << "Angle = " << angle;
    ...
  }
  catch(string& str){
    cerr << str;
  }
  catch(int i){
    cerr << "Integer: " << i << " thrown";
  }
  catch(exception& e){
    cerr << "something derived from exception";
  }
  cerr << "What did you see?" << endl;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Observer Pattern

Exceptions

- ► Introduction
- ► Syntax

Control Flow

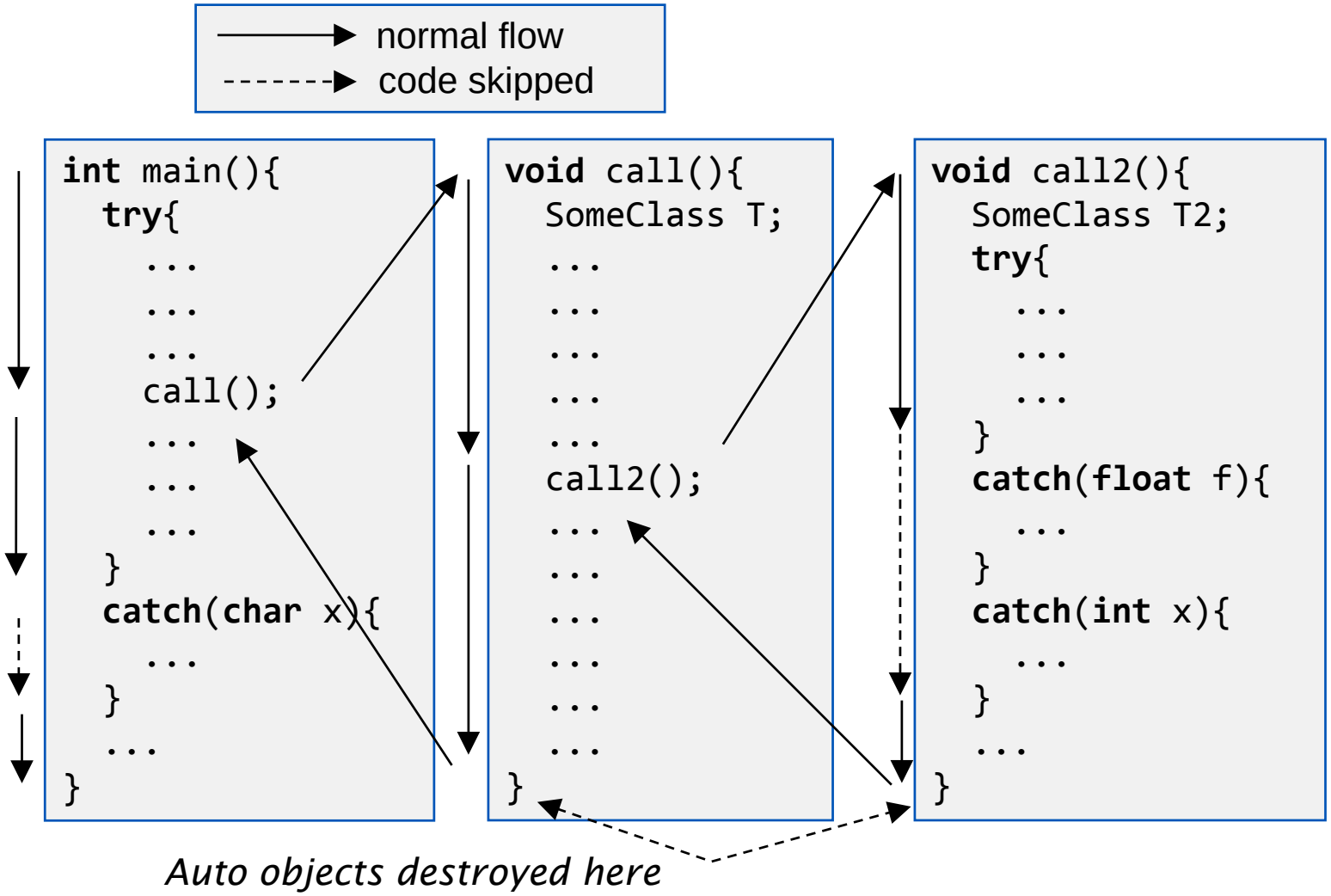
- ► Effective Exception Handling

► **Runtime Type Information**



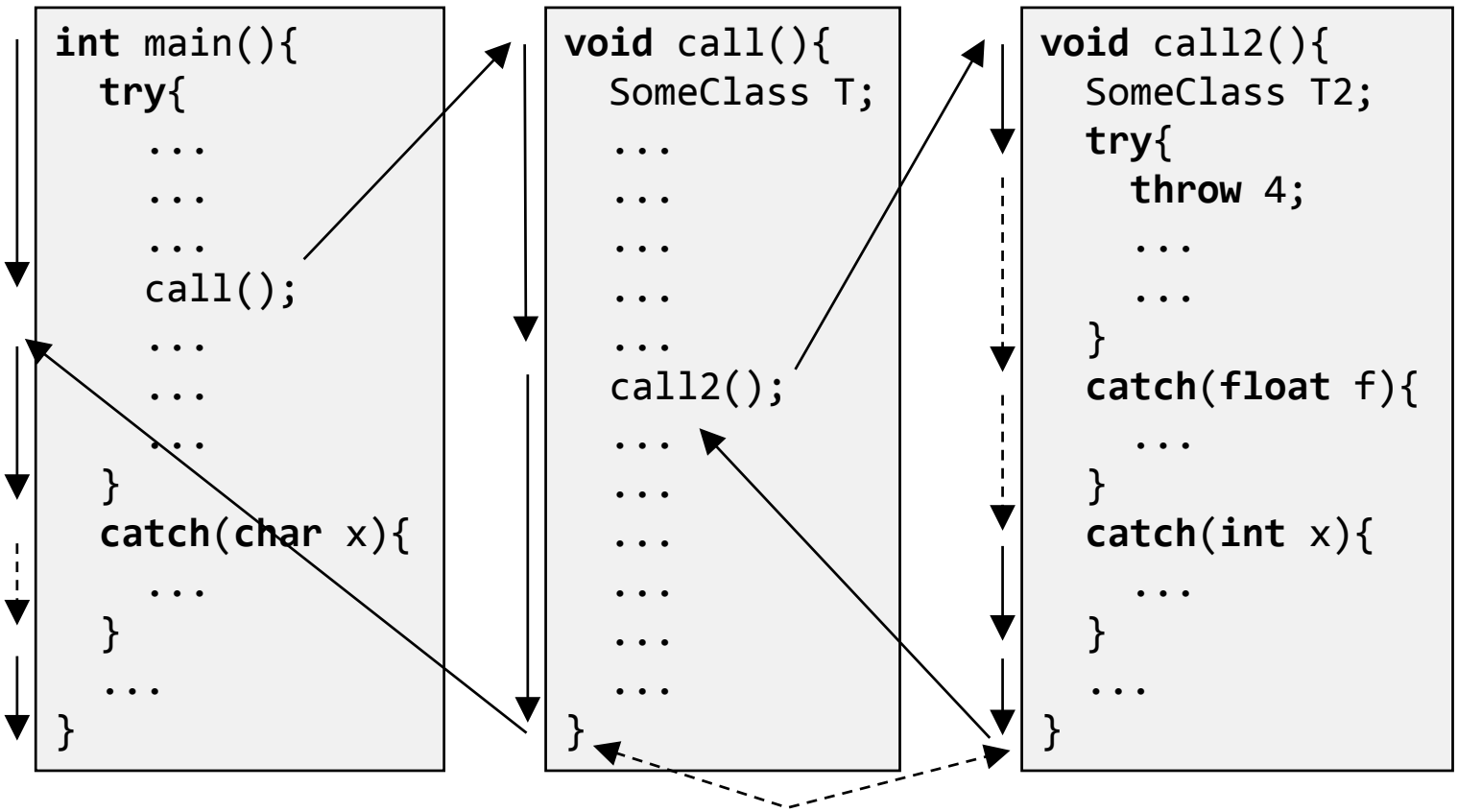
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Normal Flow



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

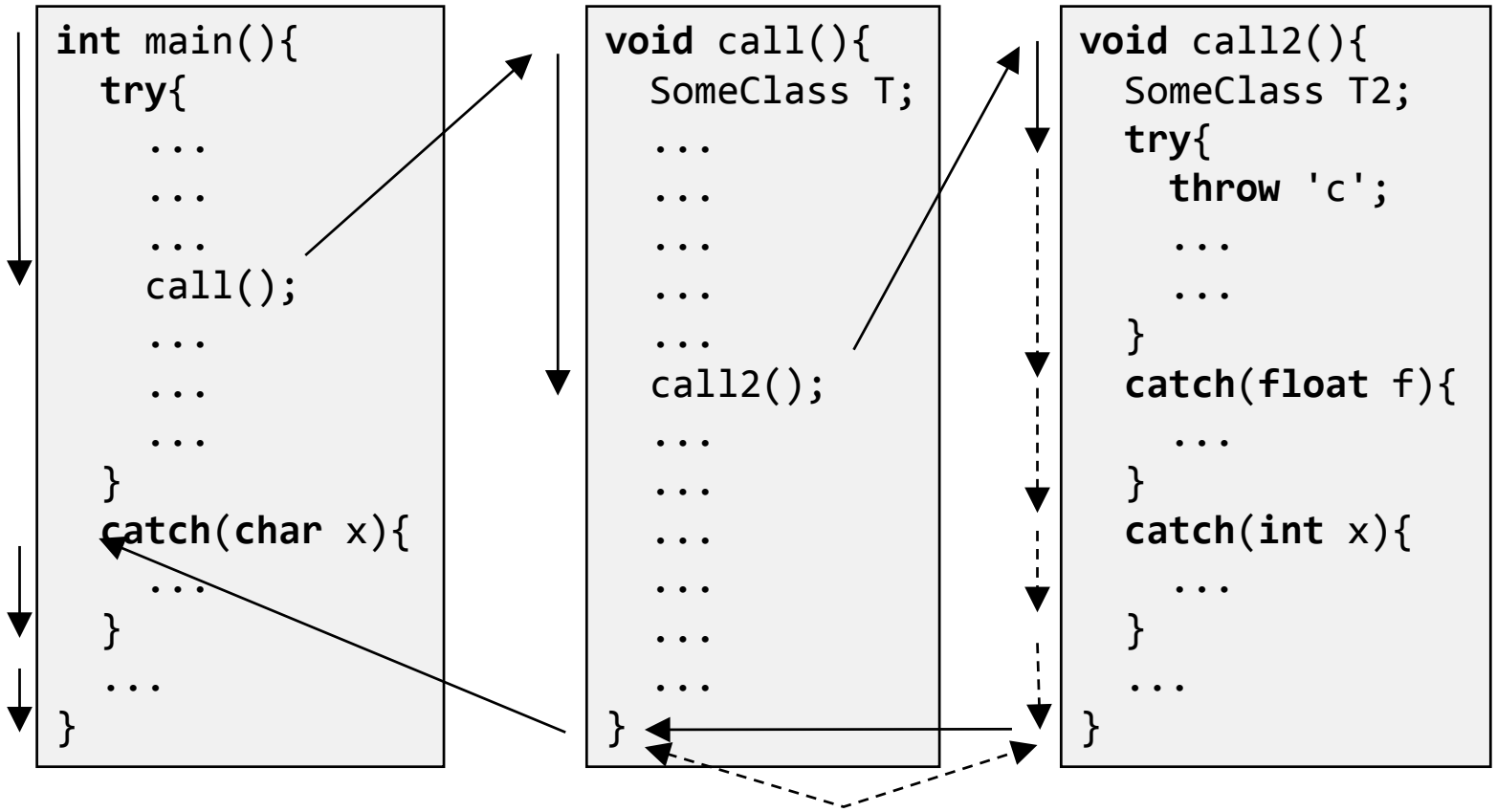
Exception Caught Locally



Auto objects destroyed here

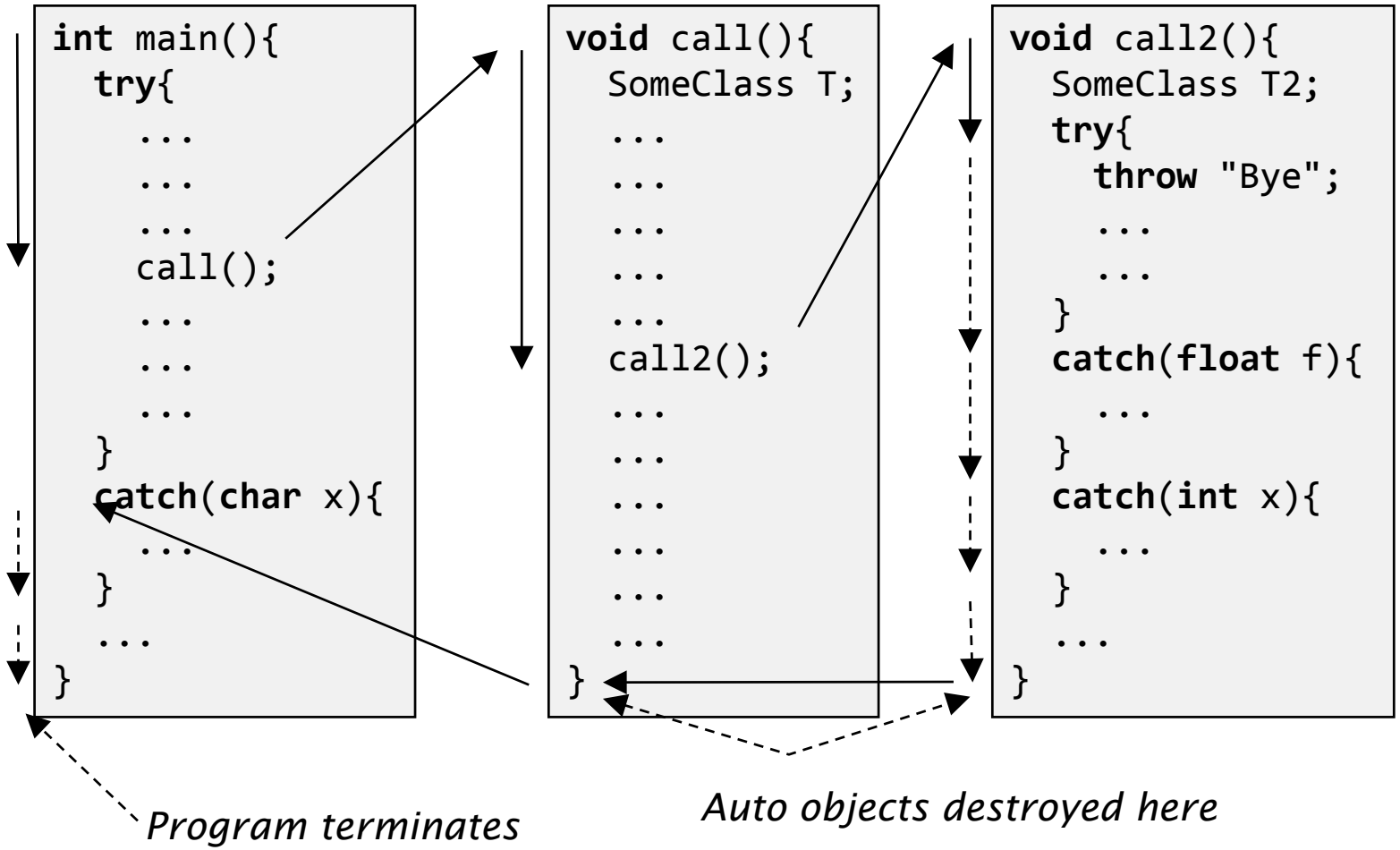
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exception Caught Remotely



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exception Not Caught



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- **Observer Pattern**

Exceptions

- Introduction
- Syntax
- Control Flow

Effective Exception Handling

- **Runtime Type Information**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exception Specifications

- **There is a way to specify that a function does not throw an exception**
  - New in C++11
  - If an exception is thrown anyway, the program will terminate

```
void doSomething1() noexcept;
```

  - The previous standard included declarations about which exceptions might possibly be thrown
    - These specifications are deprecated in C++11



```
void doSomething2() throw(DBException, RemException);
void doSomething3() throw();  // same as noexcept
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Standard Exception Class

- **The Standard Library defines the class `std::exception`**
  - All exceptions thrown by the Standard Library are of classes derived from `std::exception`
  - Best practice is that user-defined exceptions should also be derived from `std::exception`

```
namespace std {

class exception {
public:
  exception() noexcept;
  exception(const exception& rhs) noexcept;
  exception& operator=(const exception& rhs) noexcept;
  virtual ~exception() noexcept;
  virtual const char *what() const noexcept;
};

}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Catching Exceptions by Reference

Always catch exceptions by reference

- **Pass-by-value “slices” a derived object to a base class object**
- **Pass-by-pointer raises the question of who should delete it**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Handling Exceptions in Destructors

- **Ensure exceptions are not propagated from destructors**
- **Destructors are invoked when**
  - An object goes out of scope, including at program termination
  - The `delete` operator is invoked on a pointer
  - Objects are destroyed as the exception-handling mechanism performs *stack unwinding*
- **Exceptions raised within a destructor invoked in either of the first two ways follow the normal rules**
  - Handled by a matching `catch` clause of an enclosing `try` block
  - Propagated if not handled

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Handling Exceptions in Destructors

- **An exception raised but not handled within a destructor invoked as part of exception handling cannot be propagated**
  - There is no calling sequence for it to propagate along
  - The program will terminate if this situation occurs
- **A destructor cannot determine why it has been invoked**
  - To ensure exception handling is allowed to take place as required, catch all possible exceptions within destructors

```
MyClass::~MyClass() noexcept {
  try{
    doSomethingThatCanGoWrong();
  }
  catch(...){
    // must handle any exception
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exceptions and Threads



- **Do not let an exception propagate from a thread**
  - Each thread must handle its own exceptions
  - The program will terminate if an exception is not caught within the thread

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- **Observer Pattern**
- **Exceptions**

Runtime Type Information

Introduction

- Determining the Type of an Object
- Casting



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Runtime Type Information (RTTI)

- **It is sometimes necessary to know the exact type of an object**
  - Polymorphism usually removes this need
    - A shape can be drawn without knowing what kind of shape it is
- **RTTI enables the type of an object to be determined at runtime**
- **RTTI allows**
  - Comparison of object types
  - Type-safe casting

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

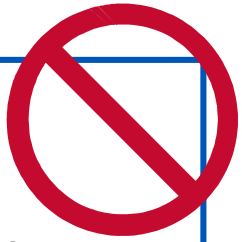
When Not to Use RTTI

- **Virtual member functions provide an elegant alternative to *switch statement from hell* programming**
  - Indiscriminate use of RTTI can bring it back

```
void f(Shape& s){
  s.draw();
}
```

rather than ...

```
void f(Shape& s){
  if(...)// s is a square
    s.drawSquare();
  else if(...)// s is a circle
    s.drawCircle();
  else if(...)// s is a line
    s.drawLine();
}
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

When to Use RTTI

- **RTTI can be used in the following situations**
  - When the base class is not available to be modified
    - So a suitable virtual function cannot be declared
  - Object data is to be written to a file
    - The type of the object must be written to allow the object to be reconstituted
    - This situation persuaded the C++ standardization committee to include RTTI in the language
  - In other situations, a solution involving polymorphism should be sought

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Observer Pattern
- Exceptions

Runtime Type Information

- Introduction

Determining the Type of an Object

- Casting



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Object Type

- **A declaration has a *static* type**
  - Type can be determined at compile time
- **An object has a *dynamic* type**
  - The runtime type of the object
- **Static and dynamic type are always equal for a simple variable definition or a parameter passed by value; e.g.:**

```
Shape s;
void f(Shape s);
```

  - `s` is a `Shape`
- **Static and dynamic type may differ for a pointer or reference**

```
Shape* p;
void f(Shape& r);
```

  - `p` and `r` may refer to a `Circle`, `Square`, or some other derived class

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Determining Object Type

- **The `typeid` operator allows the dynamic type of an object to be determined at runtime**
  - Can be applied to the target of a reference or pointer
- **The `typeid` operator can also be applied to a class**
  - In each case, **`typeid`** returns a `type_info` object
  - This allows the type of an object to be compared with that of another object or a named class

```
void f(Shape* ps, Shape& rs){
  if(typeid(*ps) == typeid(Circle)){
    cerr << "It's a circle";
  }
  if(typeid(rs) == typeid(Square)){
    cerr << "It's a square";
  }
}
```

- **It is quite rare that you need to know the *exact* class of an object**
  - Rather, the type is almost always tested with a **`dynamic_cast`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Polymorphic Types

- **Dynamic type can be determined only for *polymorphic* types**
  - A class with at least one virtual function declared
  - The virtual function pointer table is used to store a pointer to the `type_info` object
- **The `typeid` operator returns the static type of a declaration of a *monomorphic* type**
  - A built-in type or a class with no virtual functions

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aside: Polymorphism and Virtual Destructors

- **A class used polymorphically (referred to by a base class pointer or reference) should always have a virtual destructor**
  - Normally, such a class will have other virtual functions; i.e., it will already be a polymorphic type
  - A virtual destructor is essential to allow proper cleanup of derived objects
    - Even in rare cases where there are no other virtual functions
- **Best practice: If a class may be inherited from, always declare the destructor virtual**
- **What happens here?**

```
void f(){
  B* pB = new D;
  delete pB;
}
```

```
class B{
};
```

```
class D: public B{
public:
  D():arr(new int[1000]){}
  ~D(){delete [] arr;}
private:
  int* arr;
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Observer Pattern
- Exceptions

Runtime Type Information

- Introduction
- Determining the Type of an Object

Casting



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Casting

- **The old C-type cast operation is deprecated in C++**
  - Named *sledgehammer cast* by Bjarne Stroustrup
  - Very little correctness checking
  - Hard to find with its simple `( type )` syntax
- **Replaced by four casts in C++**
  - Checks that they are applied correctly
  - Have names that state what they do:
    - **`dynamic_cast`**
    - **`static_cast`**
    - **`const_cast`**
    - **`reinterpret_cast`**
  - Easy to search for them

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Dynamic Casts

- **`dynamic_cast<`*`type-id`*`>(`*`expression`*`)`**
  - Converts *`expression`* to type *`type-id`*
  - *`expression`* and *`type-id`* must both be pointers or both be references
  - Succeeds if *`type-id`* denotes the same class or an accessible base class of the dynamic type of the object denoted by *`expression`*
    - Therefore, *`type-id`* must be the same class or a class derived from the static type of *`expression`*
  - Action in case of failure depends on whether a pointer or a reference is being cast
- **`dynamic_cast` is used for type-safe *downcasting***
- **It uses the same mechanism as `typeid` to figure out whether the cast should succeed**
  - However, **`typeid`** comparison gives exact match, while **`dynamic_cast`** also succeeds for all subclasses

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Dynamic Casting of a Pointer

- **`dynamic_cast` returns a pointer to the required type if *`type-id`* denotes the same class or an accessible base class of the dynamic type of the object denoted by *`expression`***
  - Otherwise returns a null pointer

```
...
void f(Shape* pS){
  Square* pSq = dynamic_cast<Square*>(pS))
  if(pSq != nullptr){
    cout << "It's a square or it's derived from Square";
    pSq->doSquareOperation();
    ...
  }
  ...
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Dynamic Casting a Reference

- **`dynamic_cast` returns a reference to the required type if *`type-id`* denotes the same class or an accessible base class of the dynamic type of the object denoted by *`expression`***
  - Otherwise throws the exception `bad_cast`
  - It might be easier to always use **`dynamic_cast`** with pointers

```
...
void f(Shape& rS){
  try{
    Square& rSq = dynamic_cast<Square&>(rS);
    cout<< "It's a square or it's derived from Square";
  }
  catch(bad_cast& c){
    cout<< "It's something else";
  }
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Static Casts

- **`static_cast<`*`type-id`*`>(`*`expression`*`)`**
  - Compiles if there is a well-defined conversion from *`expression`* to *`type-id`*
  - Compiler must know how to do the conversion; there is no runtime checking

```
float f = 32.9;
int i = static_cast<int>(f);
```

```
class X{
public:
  X(float f):val(f){}
  explicit operator float(){return val;} // type-conversion
  explicit operator explicit int(){return val;} // type-conversion
private:
  float val;
};
```

```
int main(){
  X myX(56.2f);
  std::cout << myX << '\n';
  std::cout << static_cast<int>(myX);
  std::cout << static_cast<float>(myX);
}
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Const Casts

- **`const_cast<type-id>(expression)`**
  - Used to remove the **`const`** attribute from `expression`
- **Checks that it *only* removes `const`**
  - `type-id` and `expression` must denote the same type
- **This cast is used to allow objects declared as `const` to be modified**
  - Or passed to parameters that require **`const`**; e.g., example below
  - Undefined behavior if the data is modified when it is not modifiable
  - **`const_cast`** also removes **`volatile`**

```
void legacy_func(char*);
```

```
const char* x = "Hello World";

legacy_func(const_cast<char*>(x));
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Example: Benefit of New Casts

- **The new casts are more specific and only do one thing**
  - Avoiding non-intended casts

```
class Base{};

class Derived : public Base{};

void f(Derived* d);

int main(){
        const Base* b = new Derived();

        f(dynamic_cast<Derived*>(b));
                  // Error, cannot change constness
        f((Derived*)b);
                  // Old-style cast
                  // Valid – but probably not intended:
                  // Does downcast and removes constness
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aside: `mutable`

- **A way to allow certain data members to be modified by `const` member functions is to declare them `mutable`**
  - Think of it as meaning: Never, ever **`const`**

```
class SomeClass{
public:
  int getValue() const{
    ++callCount;
    return value;
  }
private:
  mutable int callCount;
  int value;
...
};
```

```
const SomeClass myObject;

...

cout << myObject.getValue();
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Reinterpret Casts

- **`reinterpret_cast<type-id>(expression)`**
  - Performs no checking
  - This is the sledgehammer cast, but not so easy to use by mistake
- **Reinterpret casts instruct the system to treat some memory as if it represents a particular type of object**
  - Can be used to map an object to a fixed memory location; e.g., if the location represents an I/O port

```
struct Port{
  int data;
  int flag;
};
```

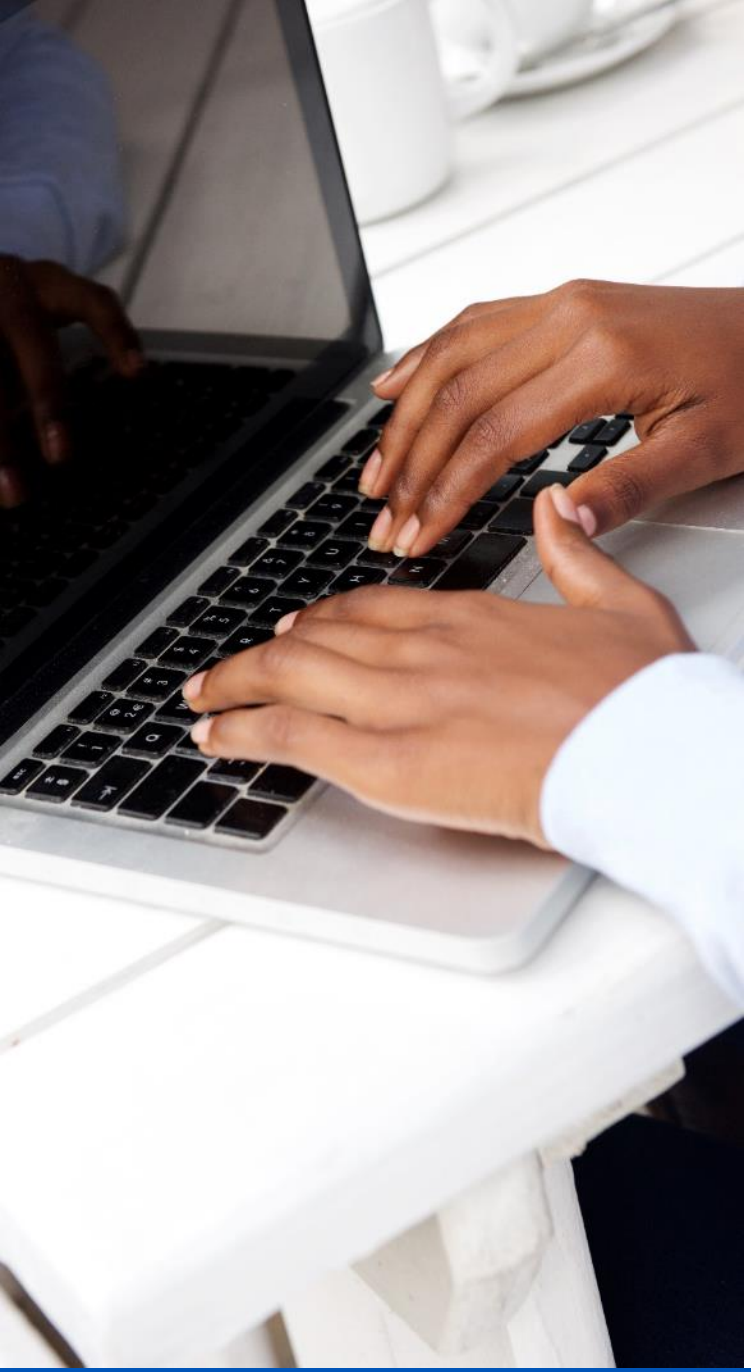
```
int getData(unsigned long address){
  Port* port = reinterpret_cast<Port*>(address);

  return port->data;
}
```

```
unsigned long myAddress = 0xffffaa00;
int x = getData(myAddress);
```

I/O = input/output

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 4.2

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 4.2: Observer Pattern Variations

In this exercise, you will be working on the pelican crossing application

You will

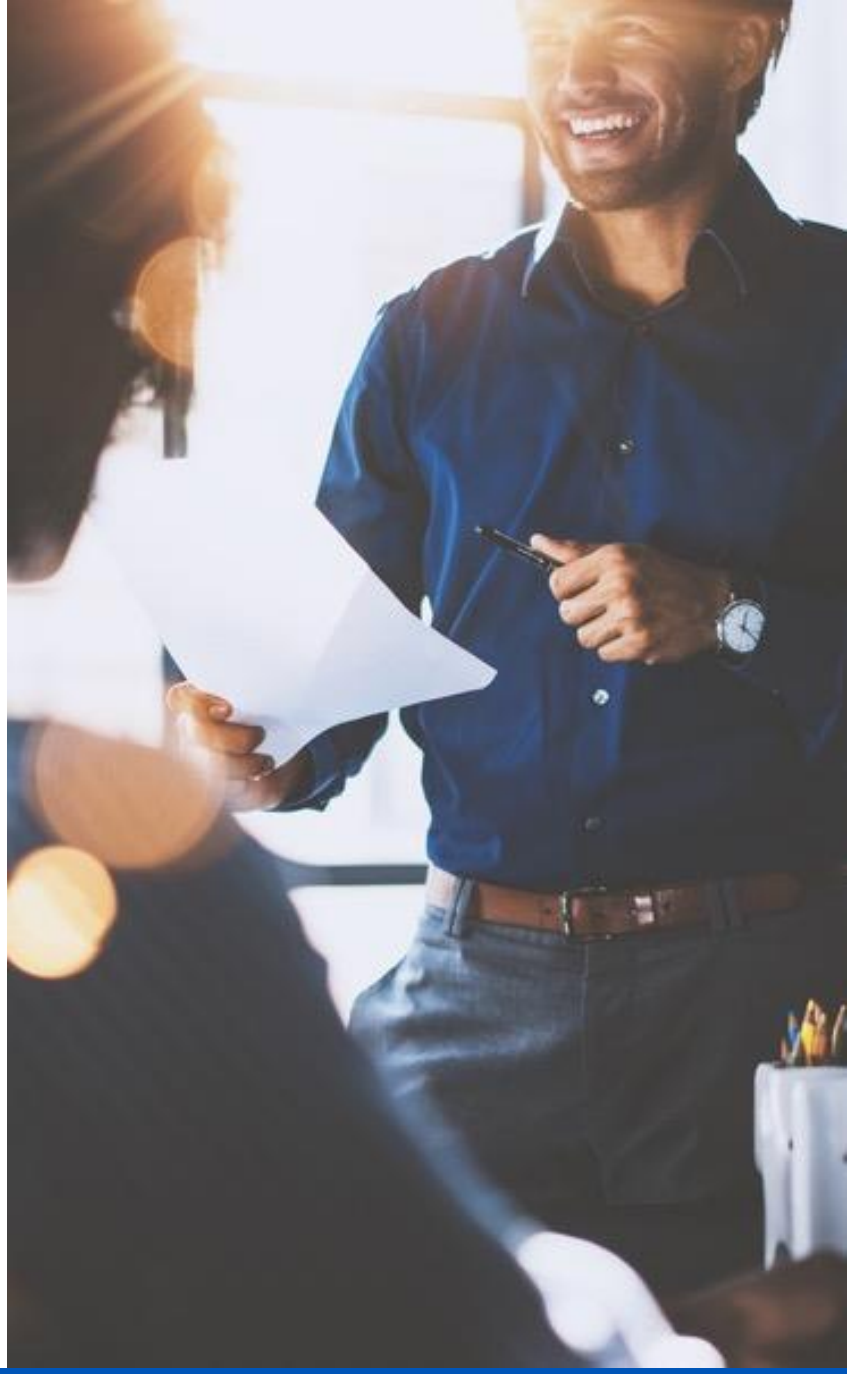
- Use RTTI in a situation where polymorphism cannot be used
- Enhance your implementation of the Observer pattern to allow the subject to tell its observers what has changed

You will also be able to experiment with various types of casts

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

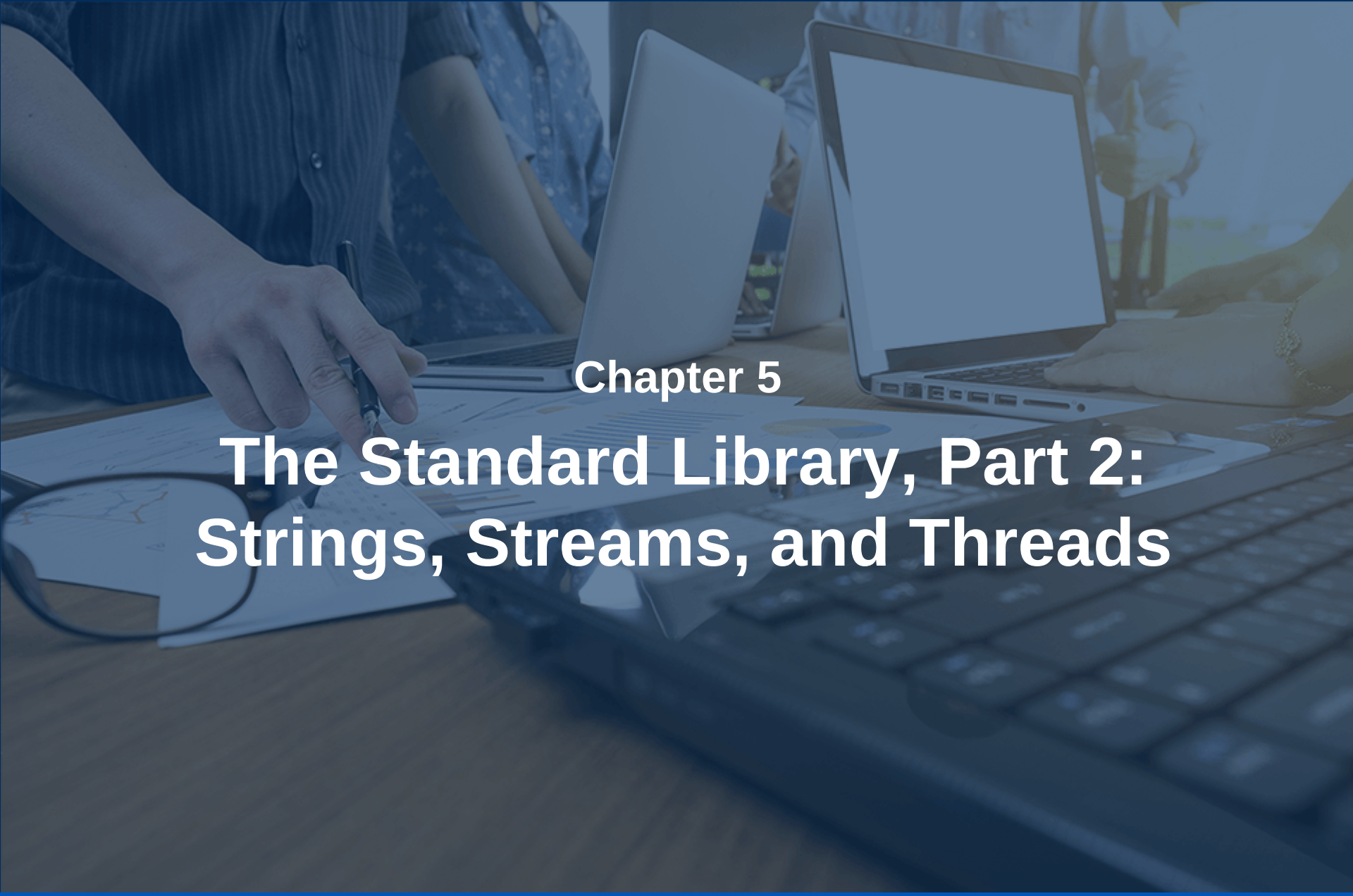
Objectives

- **Learn how the Observer pattern works**
  - Why it is prevalent in modern GUI system designs
- **Learn how C++ features are used in variations of the Observer pattern**
  - Exceptions
  - Runtime type information
  - Casts



GUI = graphical user interface

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



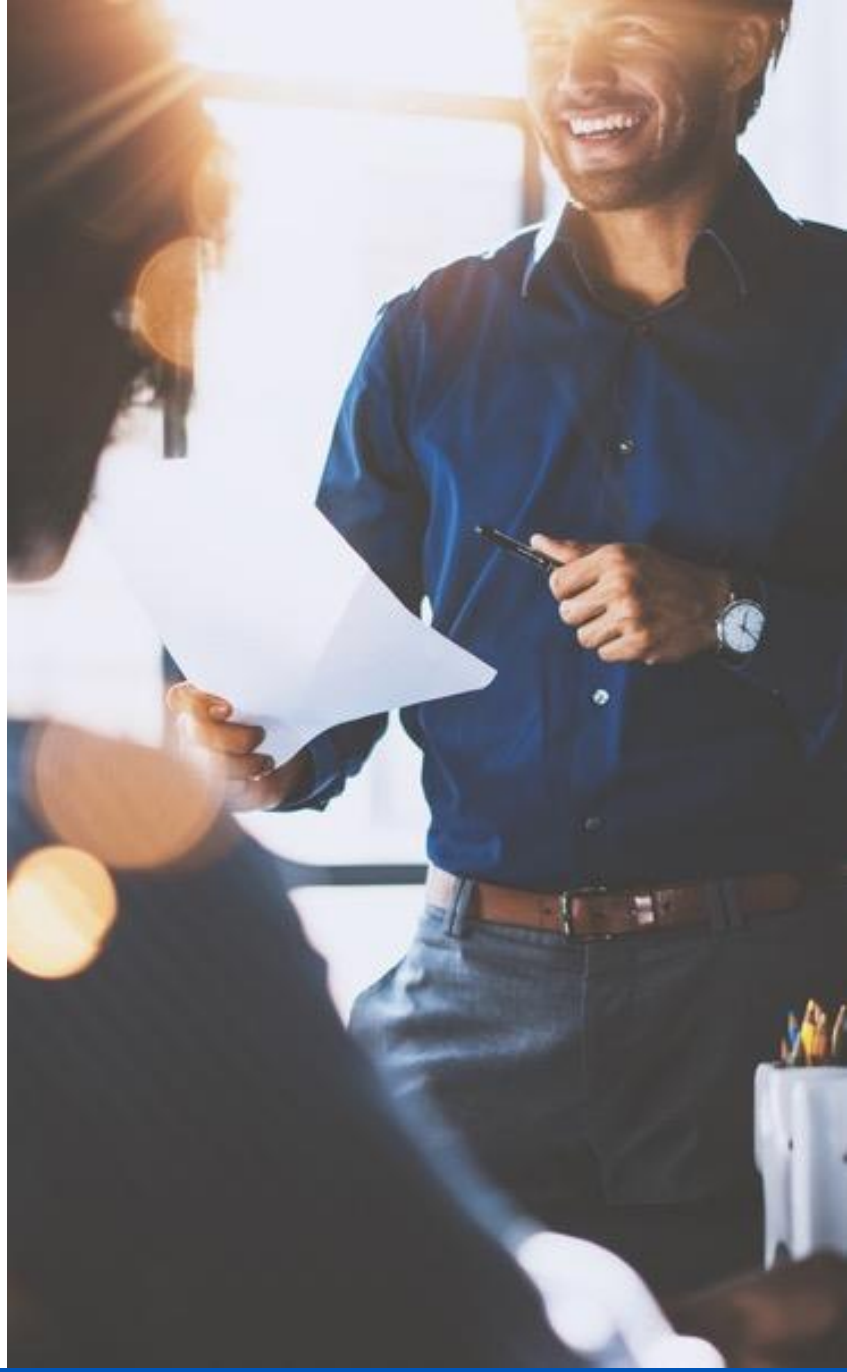
Chapter 5

The Standard Library, Part 2: Strings, Streams, and Threads

LEARNING TREE INTERNATIONAL

Objectives

- Use the Standard Library's string facilities
- Use the Standard Library's stream facilities to provide high-quality textual input and output
- Use *stream manipulators* to format textual output
- Use the error-handling facilities to defend against errors in input data
- Handle international differences
- Use the power of parallel thread execution

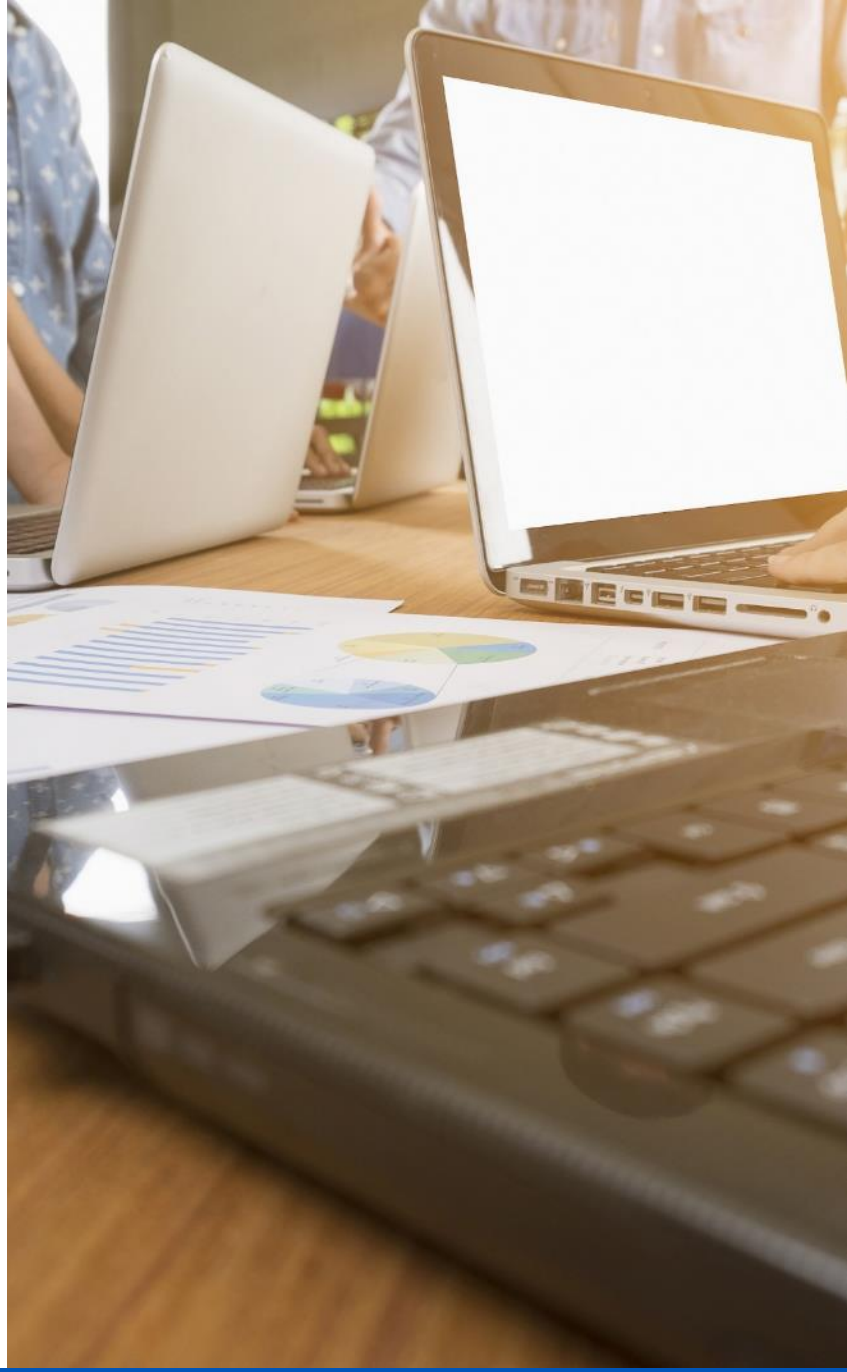


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

String Facilities

- Stream Facilities
- Internationalization
- Threads



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The String Library

- **The string facilities of the Standard Library replace proprietary libraries and the C standard string library**
  - Based on the class template `basic_string<class T>`, where `T` is the type of character used
- **The classes we use are instantiations supplied as `typedef`s**
  - `typedef basic_string<char> string;`
  - `typedef basic_string<wchar_t> wstring;`
  - `wstring` is used for 16-bit characters, used with Unicode
- **Implementations of `basic_string` can use caching, reference counting, etc., to increase efficiency**
  - So the standard string library tends to be more efficient in both space and speed than libraries based on `char*`
  - You don't have to handle memory and terminating null-bytes

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Organization of the String Library

- **Header `<string>` exports the new standard string library**
  - Based on template class `basic_string`
  - Declarations are scoped in `namespace std`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Using the `string` Library

- **Almost all uses of `char*`, `char[]`, and associated functions can be replaced by the standard `string` library**
  - Resulting code is generally more readable, more reliable, and often more efficient
  - The old C-style strings are very error-prone and dangerous
- **The C++ `string` library has a full set of string operations**

```
// Old C-style strings
#include <cstring>
using namespace std;


int main(){
  char str1[12]="Hello ";
  char str2[6];
  strcpy(str2, "World");
  strcat(str1, str2);
  cout << str1;
}
```



```
//C++
#include <string>
#include <iostream>
using namespace std;

int main(){
  string str1="Hello ";
  string str2;
  str2 = "World";
  str1 += str2;
  cout << str1;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Compatibility With Legacy Code

- **To use a library that accepts only `char*` style strings, use the member function `const char* string::c_str();`**
  - Returns a C-style, null-terminated string pointer
  - The pointer becomes invalid if the C++ string changes
    - Don't save the **`const char*`** pointer!

```
...
void legacyFunc(const char* str){
  ...
}
```

```
...
int main(){
  string str1="Hello ";
  legacyFunc(str1.c_str());

  const char* ptr = str1.c_str();
  str1 += "World";
  legacyFunc(ptr);
}
```

Correct ----------▶ `legacyFunc(str1.c_str());`

Bad practice ----------▶ `const char* ptr = str1.c_str();`

Undefined behavior ----------▶ `legacyFunc(ptr);`



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

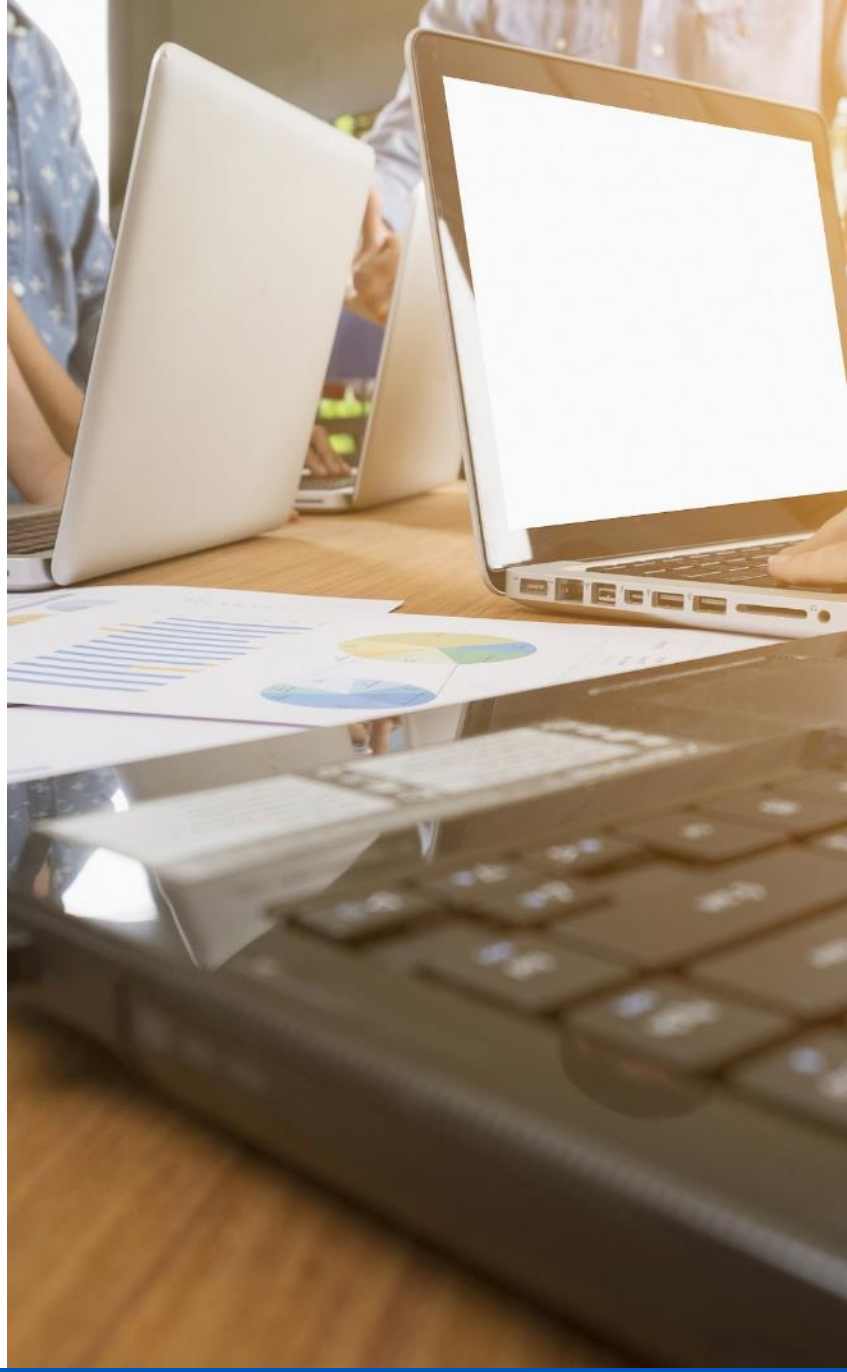
- **String Facilities**

Stream Facilities

Organization

- Simple Input/Output
- Handling Input

- **Internationalization**
- **Threads**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Library Organization

- **The stream facilities are declared in headers `<iostream>`, `<fstream>`, and `<sstream>`**
  - Provides equivalent functionality to `<stdio.h>`
- **`<cstdio>` declares the C Standard Library**
  - For example, `printf()`, `scanf()`, `sprintf()`, `fscanf()`
  - Wrapped in `namespace std`
  - `stdio.h` is deprecated

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

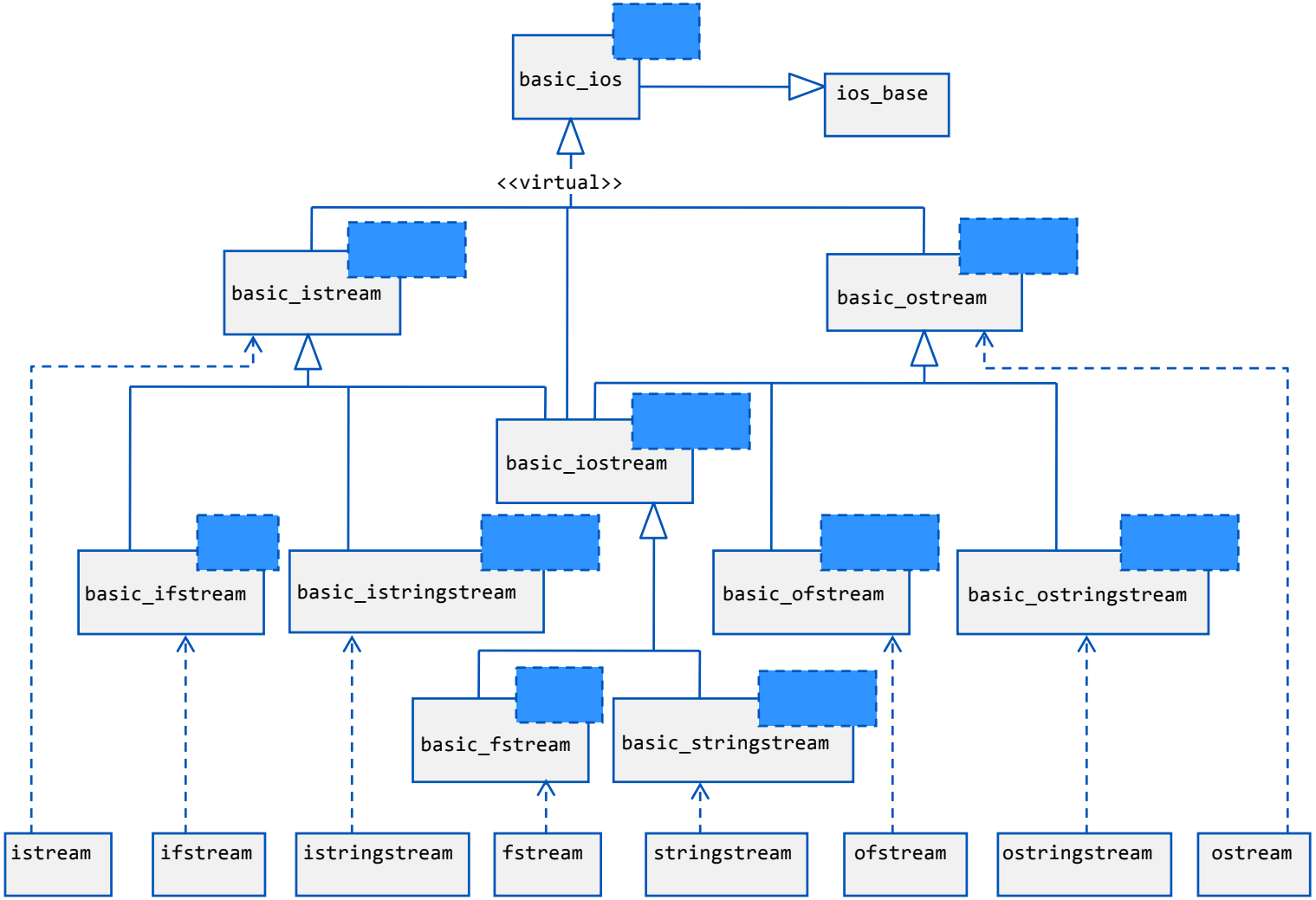
Streams

- **A *stream* is used as a source or sink for characters**
- **Stream templates provide functionality for combinations of**
  - Input, output, or both
  - To/from file, standard input/output device, or in-memory strings
- **Streams for handling `char` are instantiations of these templates for type `char`, for example**
  - The stream for outputting `chars` to a file is defined as

```
typedef basic_ofstream<char> ofstream;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Stream Hierarchy



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Predefined Stream Objects

- **The following stream objects are declared in `<iostream>`, wrapped in `namespace std`**

`istream cin;`	standard input stream
`ostream cout;`	standard output stream
`ostream cerr;`	standard error stream
`ostream clog;`	standard logging stream

- **These streams can be assigned to files from outside of the program**
  - Text-processing programs often use these objects rather than explicit files for flexibility
- **Wide-character versions exist**
  - `wistream wcin;`
  - `wostream wcout;`
  - Etc.
  - These were meant to be used for 16-bit Unicode characters, which have almost totally been replaced with UTF-8 encoding of Unicode

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating Stream Objects

- **Stream objects can be created like any other type of object**

```
#include <fstream>
using namespace std;

ofstream file("data.dat");
...
```

New in C++11:
File stream constructor would also accept `string` parameter, previously only ***`const char*`***

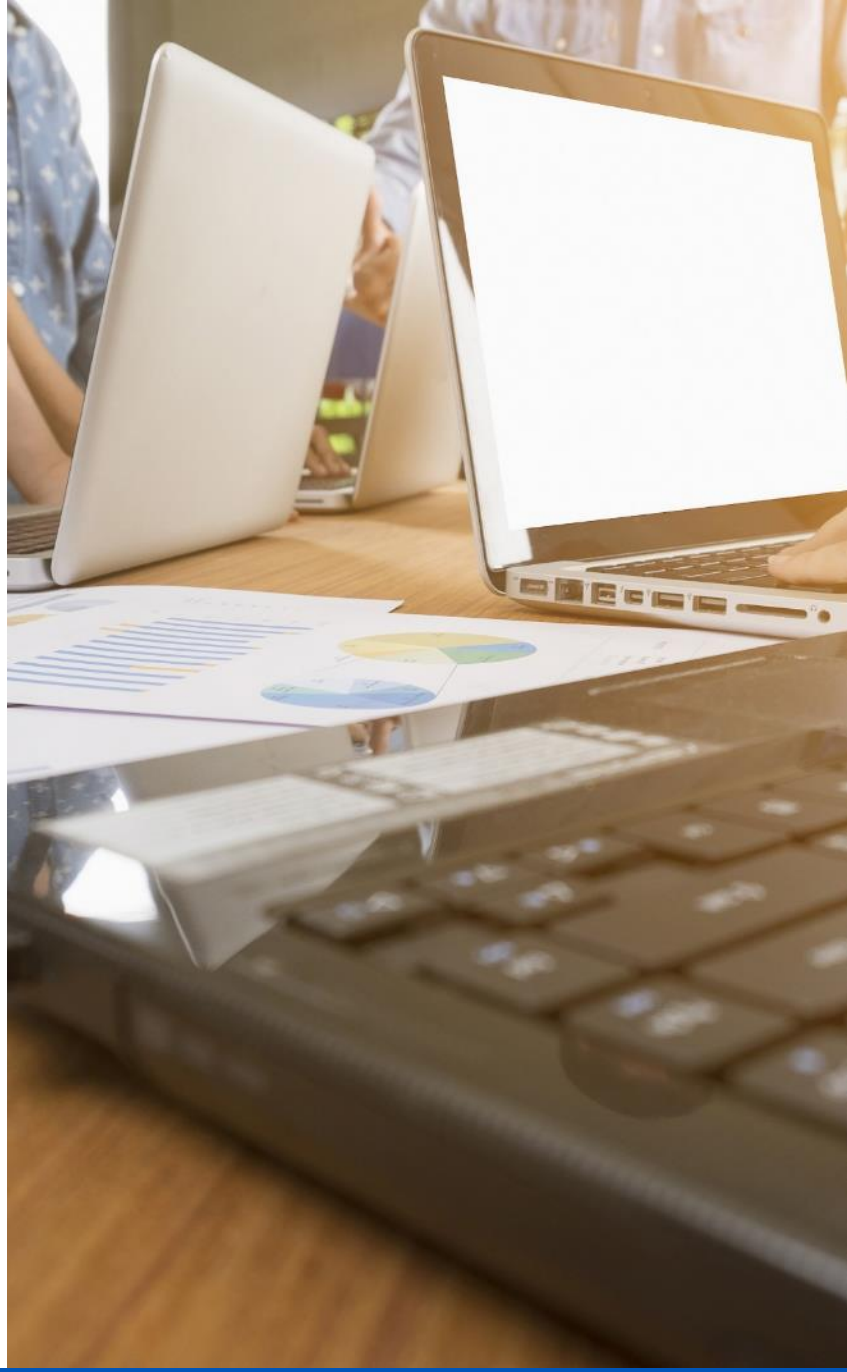
- **Streams created like this are accessed in the same way as the predefined streams, `cout`, etc.**
  - The class of the stream determines the operations available; there is nothing special about `cout`

```
file << "Hello world";
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- **String Facilities**
- **Stream Facilities**
  - Organization
  - **Simple Input/Output**
    - Handling Input
- **Internationalization**
- **Threads**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Insertion Operator

- **Output to a stream is done using the *insertion operator*, `operator<<`**
- **Overloaded for all built-in types, plus the string type appropriate to the character type in use**
  - `string` for type `char`; `wstring` for type `wchar_t`
- **Many stream manipulators exist to format output data**
  - For how long do these influence formatting?

```
#include <iostream>
using namespace std;

int main(){
  int value = 45;
  cout << "Hello World\n";
  cout << hex << value;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Defining an Insertion Operator

- **To provide `operator<<` for a user-defined class, you must define a global function with two parameters**
  - Since the stream object is on the left-hand side, a member function would have to be defined in the `ostream` class, which we don't have access to
- **General form for an insertion operator is:**

```
ostream& operator<<(ostream& os, const X& x){
  os << x.getData() << x.getMoreData(); // etc.
  return os;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Insertion Operator: A Tip to Remember

- **Insertion `operator<<` has to be a global function, not a member**
  - Thus, it will have no private access to the class it belongs to
- **Instead of declaring it a `friend`, define a member function `print`**
  - Call it from **`operator`**`<<`
  - Makes it possible to make `print` polymorphic, which **`operator`**`<<` cannot be

```
class X {
        ...
  public:
     virtual ostream& print(ostream& os = cout) const {
        ...
        return os;
     }
     ...
}

ostream& operator<<(ostream& os, const X& x){
        return x.print(os);
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Extraction Operator

- Input from a stream is done using the *extraction operator*, `operator>>`
- Overloaded for all built-in types, plus the string type appropriate to the character type in use
- For example, here is a user-friendly program:

```
#include <iostream>
using namespace std;

int main(){
  cout << "Enter your age: ";
  int x;
  cin >> x;
  cout << "Wow, you don't look a day over " << x - 4;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

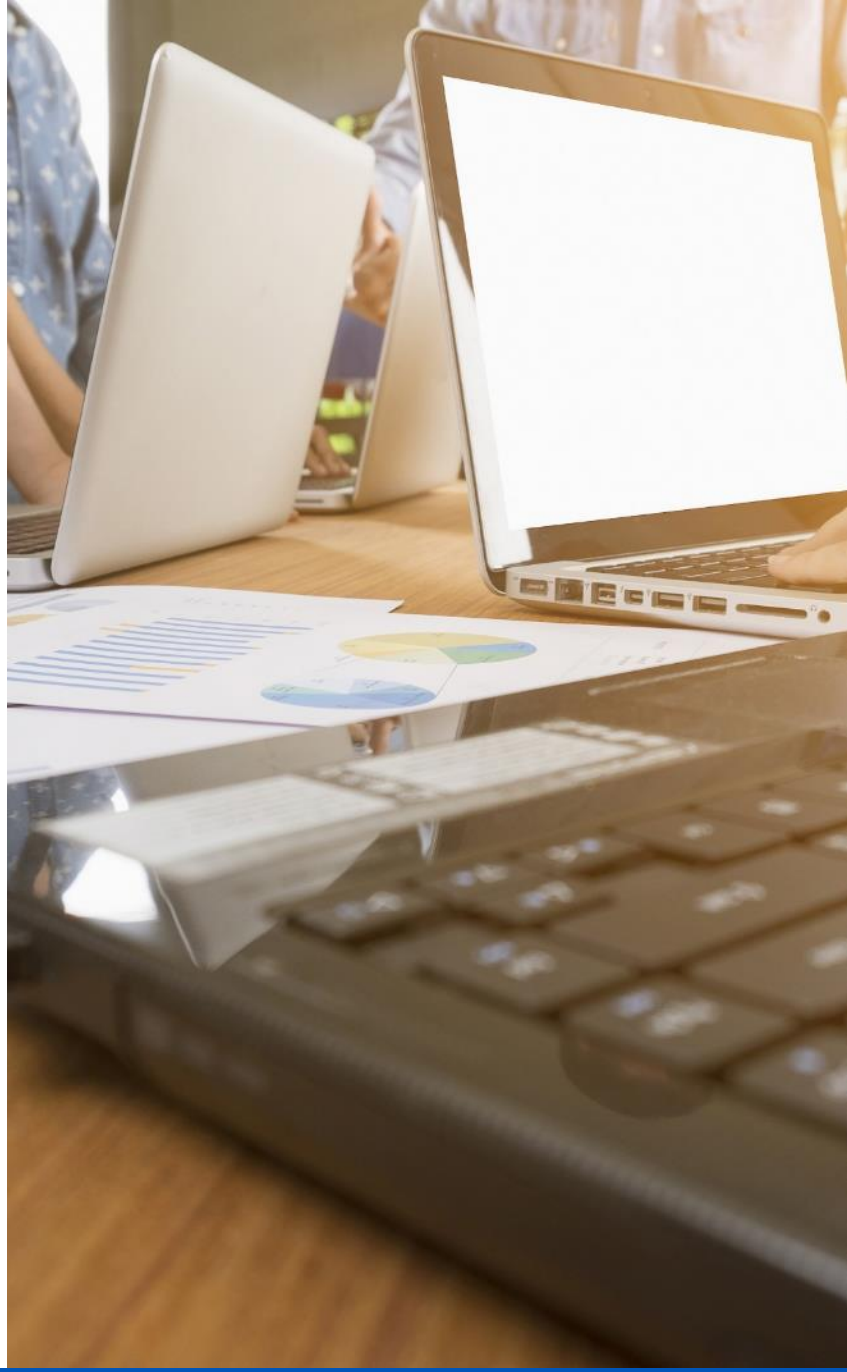
- String Facilities

Stream Facilities

- Organization
- Simple Input/Output

Handling Input

- Internationalization
- Threads



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Handling Input Errors

- **When an input error is detected, the stream's *fail flag* is set**
  - This can be detected using member function `fail()` or `operator!()`
- **All further input will fail until the flag is reset using member function `clear()`**

```
int x;

cin >> x;

if(!cin){      // or could use cin.fail()
  cin.clear(); // only resets the error flag!
  string str;
  cin >> str;
  cout << "Expected an int, got: " << str << endl;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

White Space and End of File

- **By default, white space is skipped on input**
  - The manipulator `noskipws` allows white space to be read
  - To revert to skipping white space, use stream manipulator `skipws`
- **To check for reaching the end of a file when extracting from a file stream, use member function `eof()`**
  - What does this code do?

What would happen if this line were removed?

```
ifstream f("text.dat");
f >> noskipws;

char c;

f >> c;
while( !f.eof() ){
  cout << c;
  f >> c;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inserting Into String Streams

- **A *string stream* allows insertion and extraction operators to be used on in-memory strings**
- **Characters may be inserted into an `ostringstream`**
  - Member function `str()` returns the characters as a string

```
#include <sstream>
...
string formatDate(int y, int m, int d){
  ostringstream oStr;

  oStr << y << ' ' << m << ' ' << d;
  return oStr.str();
}
```

```
int main(){
  string date = formatDate(58, 8, 24);
  cout << date;
}
```

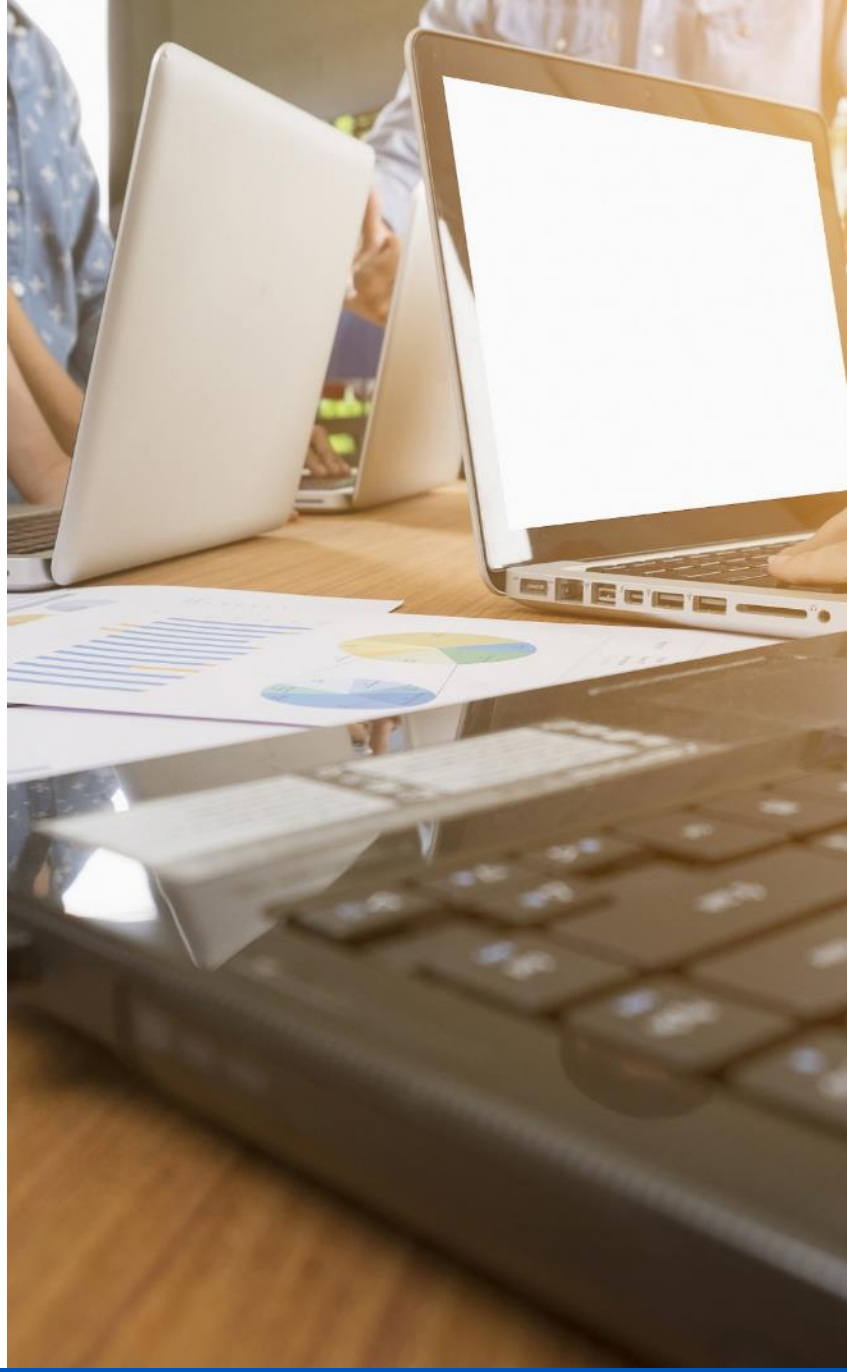
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- String Facilities
- Stream Facilities

Internationalization

- Threads



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Character Types and Traits

- **The Standard Library handles different character sets by allowing both the character type and the way characters are interpreted to be varied**
- **Most classes that act on characters are templates taking two parameters**
  - The character *type*, which defines the values a character can take
  - Character *traits*, which define the way characters are handled; e.g.:
    - What is a capital letter?
    - What is the sorting order?

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Standard Character Types

- **Type `char` is 8 bits wide**
  - "Ordinary" strings and streams are instantiations of Standard Library templates for type **`char`**; e.g.:
    - `string`
    - `ifstream`
    - `ostream`
- **Type `wchar_t` is (at least) 16 bits wide**
  - Designed for holding 16-bit Unicode characters, now more or less obsolete
  - Wide strings and streams are instantiations of Standard Library templates for type **`wchar_t`**; e.g.:
    - `wstring`
    - `wifstream`
    - `wostream`
- **Unfortunately, `wchar_t` does not fit all Unicode characters, since they can be larger than 16 bits**
    - The most common solution is to use UTF-8 (variable size) encoding of Unicode and use single-byte **`char`**s to store them

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Locales

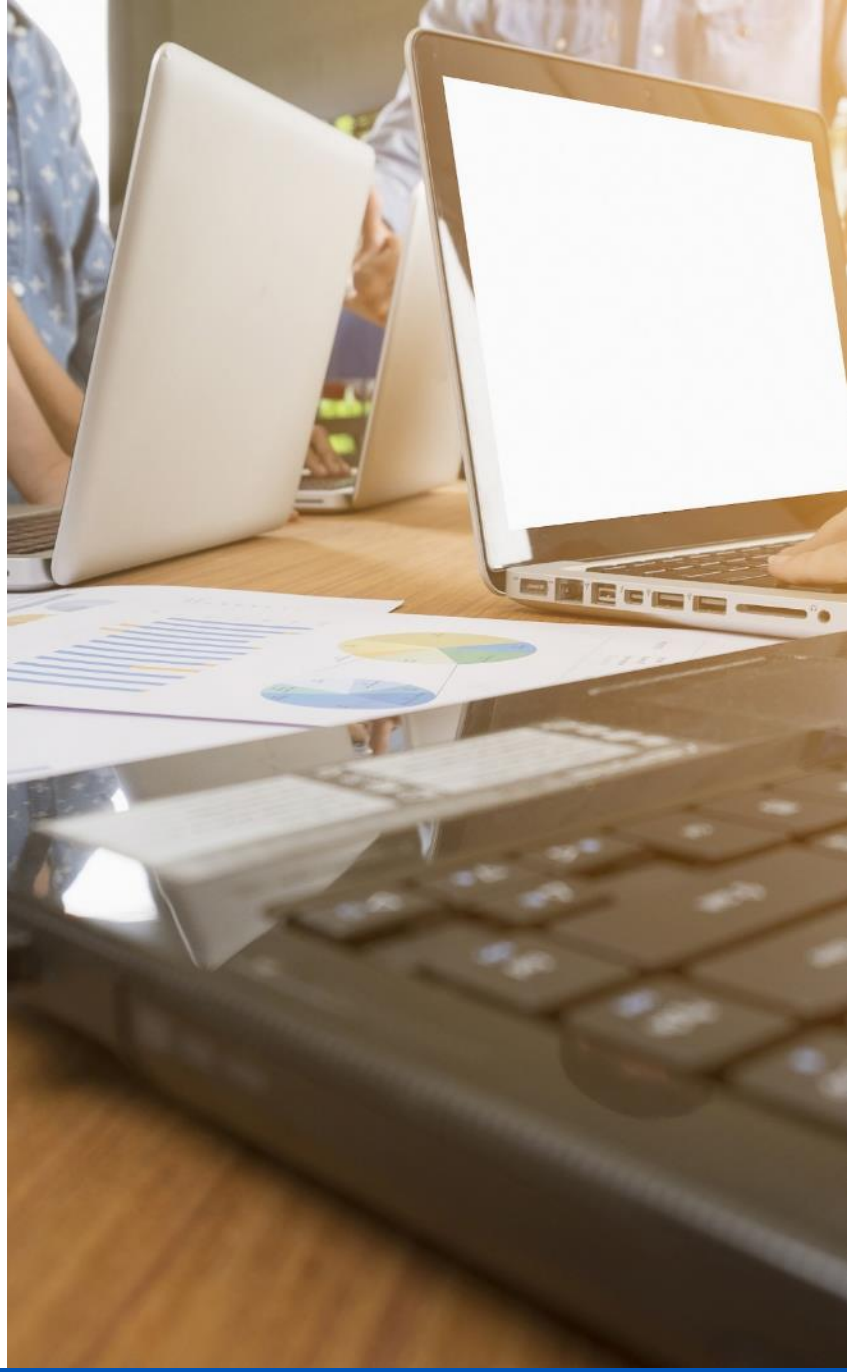
- **Local differences are handled by `locales`**
  - Vary per country and language
- **A `locale` object specifies a set of `facets`**
- **A `facet` specifies how a particular aspect of the local language or conventions should be handled; e.g.:**
  - What is the currency symbol?
  - How are numeric strings to be punctuated?
  - How are dates expressed?
- **A program can be made to behave correctly for different cultures by supplying an appropriately constructed `locale` object**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- String Facilities
- Stream Facilities
- Internationalization

Threads



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Threads

- **The *thread library* is new in C++11**
  - Added as a standard interface specification
  - The library maps to the existing thread mechanism of the host operating system
  - Code portability between different operating systems is the benefit
- **A thread is a separate execution environment within the same program**
  - A program seems to be running in more than one place at the same time
  - It can be truly parallel if there are many processors
- **A thread has its own stack**
  - Local variables are thus not shared by default
  - All of the program memory can be accessed by all threads
  - Great care has to be taken when sharing data
  - Portions of code that access the same data must almost always be mutually excluded not to run at the same time, unless everyone is only reading the data

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Threads Are Tricky

- **Thread programming is a very tricky concept**
  - This course section is a short introduction
- **General advice for using threads**
  - Don’t use them
  - If you do, don’t share anything between the threads, even indirectly
  - If you do, share minimal data and protect access to it carefully
  - Never hope that ad hoc threads will work
    - They may seem to work fine for a long time if there happen to be no collisions in accessing shared data
    - They may even pass all tests, all by pure luck
- **When will your pure luck expire?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Thread Usage Scenarios

- **Threads can divide work between processors**
  - We have a heavy calculation to do; e.g., signal processing of a large sound clip
  - Divide the clip into two parts
  - Let two threads each process half of the clip
  - No sharing of data; hopefully about twice as fast if we have two processors
- **Write code for separate actions separately**
  - It is hard and usually ugly to write code that handles many separate things; e.g.:
    - Waiting for keyboard input *and* waiting for a timer to expire
    - Better to code them as separate actions and execute each in a separate thread
- **Very often, a separate thread is waiting for something; e.g., data from the network, and thus does not need a processor to run it**
  - You can't "wait faster" with a faster processor
  - But many threads can wait in parallel

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Thread Coding Example

- **The code to execute is packaged as a function**
  - Then given to a thread object for execution
  - Often, the thread creator waits for the thread to finish its work

```
#include <thread>
using namespace std;

void myThreadFunc() {
  doSomethingA();
}

int main() {
  cout << "start thread";
  thread t1(myThreadFunc);
  cout << "thread started";
  doSomethingB();
  t1.join();
}
```

Wait for thread `t1` *to finish its work; i.e., return from its thread function*

- **Which will start executing first, `doSomethingA` or `B`?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Thread Coding Example

- **The argument to the thread may be anything that can be called**
  - A function pointer
  - A function object, implementing **`operator`**`()()`
  - A lambda expression
- **Sometimes, a creator of a thread does not want to wait for it to finish**
  - Then the thread should be `detach`ed so it can die on its own
  - It will be killed when program exits
- **Is it OK for a thread to be killed in the middle of its execution?**

```
int main() {
  cout << "start thread";
  thread t1([](){doSomethingA();});
  t1.detach();
  cout << "thread started";
  doSomethingB();
}
```

The `t1` *thread will be killed when* `main` *exits*

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Thread Locking Example

- **What if more than one thread uses the same resource or data?**
  - Assume that both `doSomethingA` and `doSomethingB` call this function

```
void log(const string& msg) {
  string s = "Log: " + msg + "\n";
  cout << s;
}
```

- **Only one thread can use the `cout` insert operator at a time. Why?**
  - We need to lock that code, for mutually exclusive access

```
std::mutex cout_mutex;

void log(const string& msg) {
  string s = "Log: " + msg + "\n";
  cout_mutex.lock();
  cout << s;
  cout_mutex.unlock();
}
```

All need to share the same lock

Lock only the critical section of code

- **What about calls to `cout` insert operators elsewhere?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Thread Locking Example

- **What happens if we forget to `unlock` the `mutex`?**
  - It could happen by mistake or by skipping the `unlock` due to an exception
- **The library has a solution for this**
  - The `lock_guard` unlocks its `mutex` in its destructor

```
std::mutex cout_mutex;

void log(const string& msg) {
  string s = "Log: " + msg + "\n";

  std::lock_guard<std::mutex> my_lock(cout_mutex);
  cout << s;

  ...
}
```

Lock section of code from here

The `unlock` *is automatic in destructor of* `my_lock` *regardless of how we got to the end of the block*

- **Can we delimit the locked section with an inner block using { }?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Threads: Sleeping and Timing

- **Since many threads will do repeated work, it is common that they need to wait or sleep**
  - C++11 has a new library section for precision timing: `<chrono>`
  - An example that will say a greeting every 10 seconds:

```
#include <chrono>


void f()
{
   std::chrono::milliseconds sleepTime(10000);
   while (true) {
        cout << "Hello\n";
        std::this_thread::sleep_for(sleepTime);
   }
}
```

- **How is time *really* measured in the operating system?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More on Lambda Expressions

- **What can lambdas do?**
  - A lambda works like an in-line function object and should be regarded as such
  - It also adds the capability of *capturing* variables from the context where it is used
    - These captured variables must be specified in the [ ] brackets
    - The variables can be captured as copies or references
    - Think of these as data members of the function object class, initialized by the constructor of the function object class

```
vector<Apple> myApples;
...
int count = 0;
bool reverse = false;
sort(myApples.begin(), myApples.end(),
     [reverse, &count](const Apple& l, const Apple& r){
                  count++;
                  return reverse ? r < l  : l < r;
});
cout << "Number of comparisons: " << count;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lambdas in Classes

- **If a lambda is in a member function and needs to use other data members and member functions, `this` needs to be captured**

```
class Worker{
private:
  std::thread runner;
  ...

public:
   Worker() : runner([this](){ this->doRepeatedly(); } ){
      ...
   }

   void doRepeatedly() {
      while (true) {
           ...
      }
   }

  ...
};
```

this->
is implicit here
and could be omitted

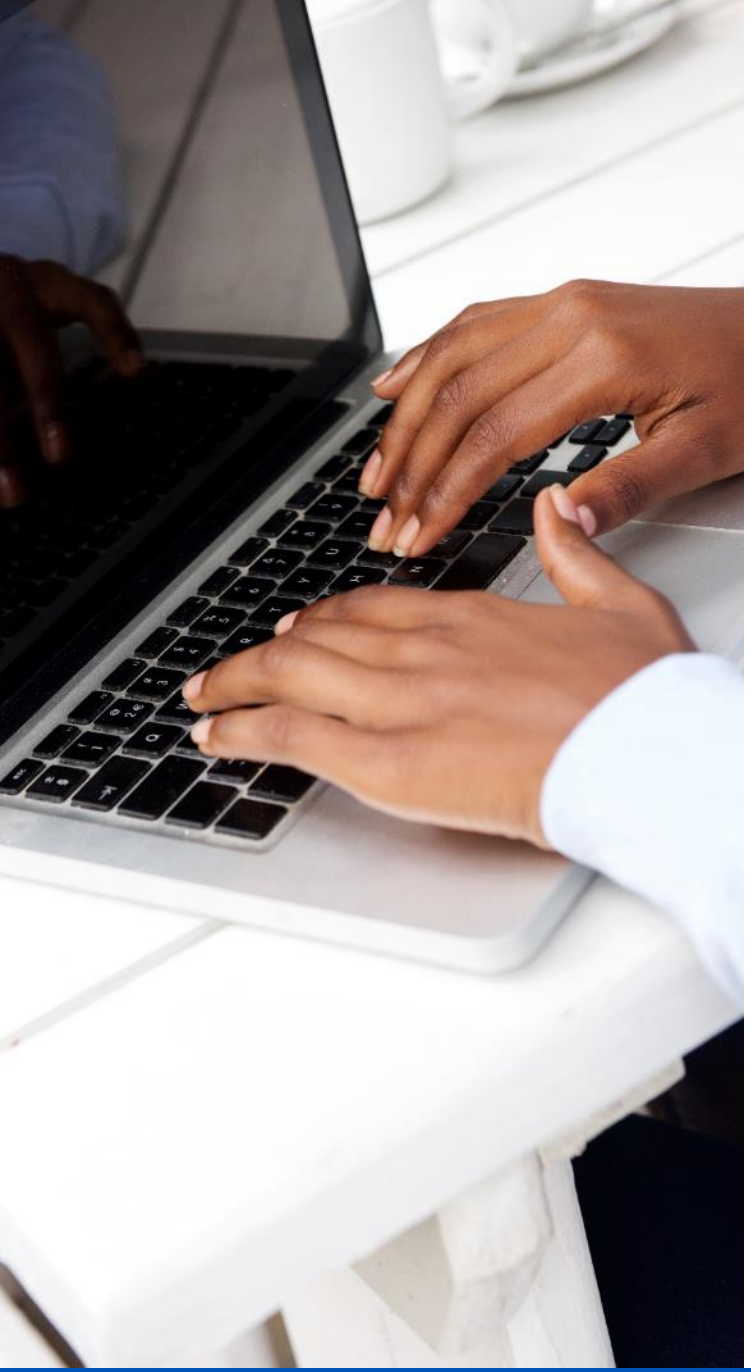
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Type of a Lambda

- **The exact type of a lambda is implementation defined**
  - What if we want to receive a lambda as a parameter?
  - There is a library template **`std::function`** that will match *anything that can be called*:
    - Function pointer
    - Function object
    - Lambda expression
  - The second argument to the function below can be any of these, as long as it takes two **`int`**s as parameters and returns **`bool`**

```
void msgAndAction(std::string msg, std::function<bool(int, int)> f) {
    cout << "Message: " << msg << endl;
    cout << "Result with 2, 3: " << f(2,3);
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 5.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 5.1: Using `std::thread`

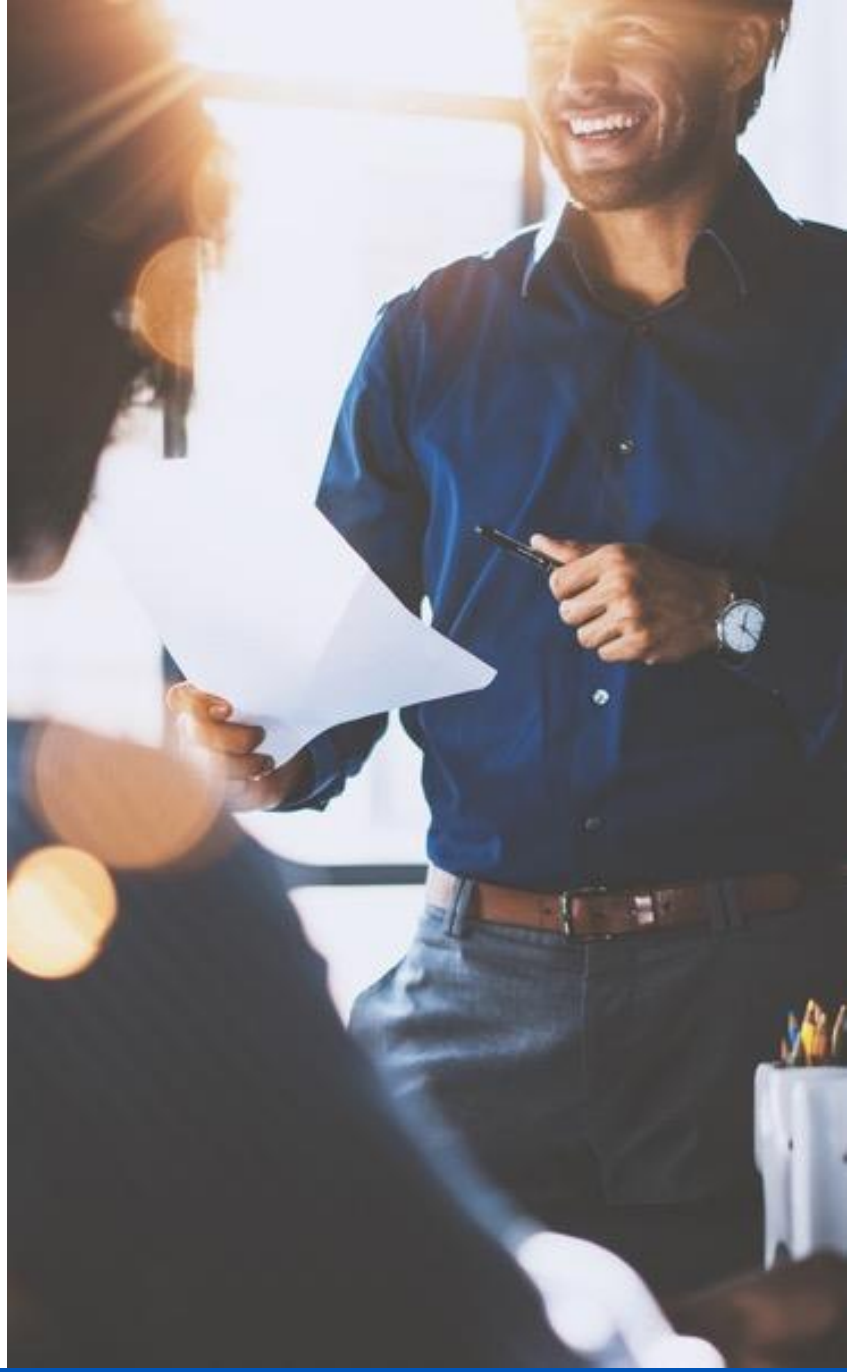
In this exercise, you will use a separate thread to poll the `TimerSet`

- So far, we have done that by calling `pollAllTimers` repeatedly from the main program—not a very pretty solution
- The `TimerSet` should handle its own polling
- It shouldn't poll at full speed as before; every 0.1 second should be enough

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Use the Standard Library's string facilities
- Use the Standard Library's stream facilities to provide high-quality textual input and output
- Use *stream manipulators* to format textual output
- Use the error-handling facilities to defend against errors in input data
- Handle international differences
- Use the power of parallel thread execution



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



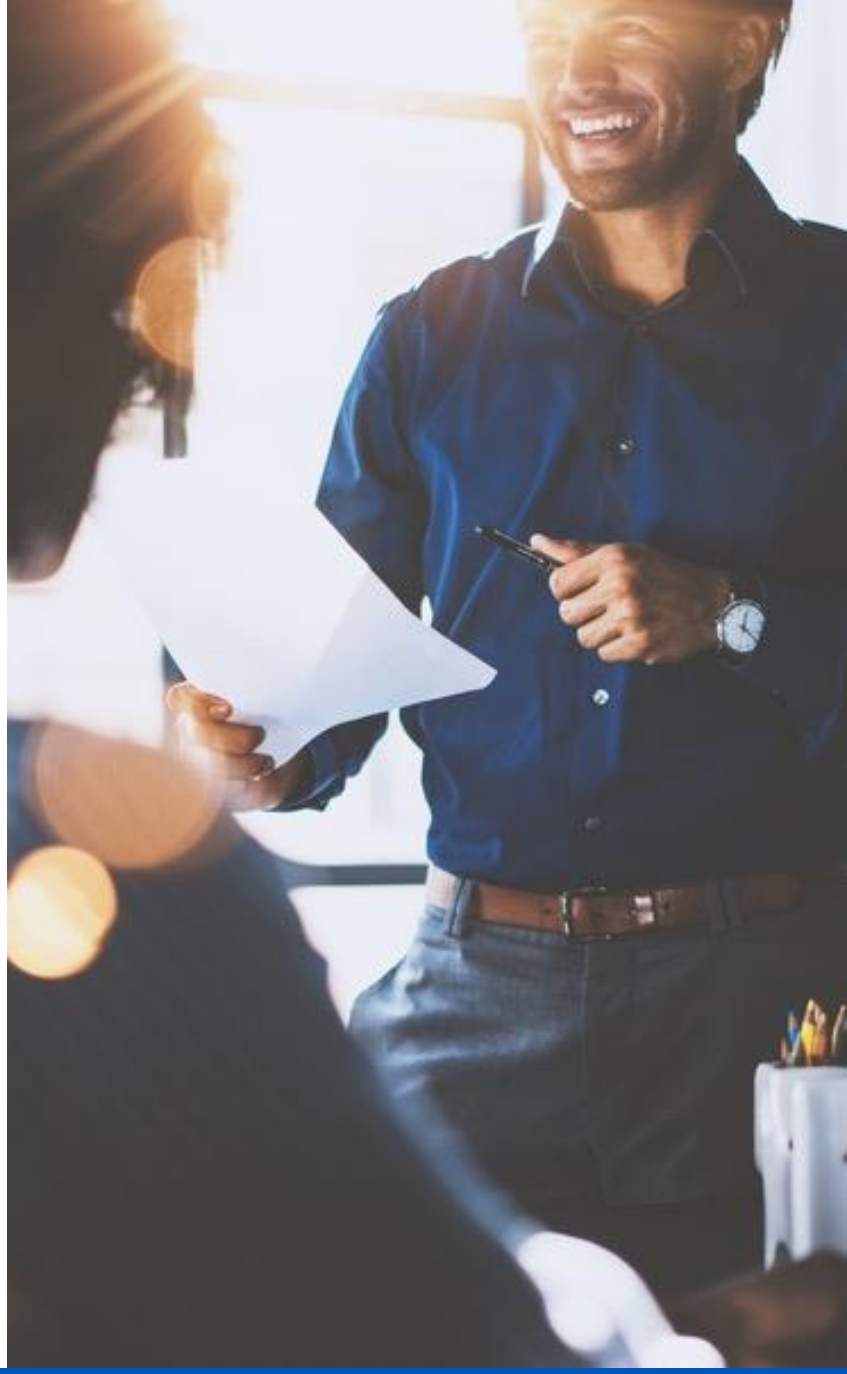
Chapter 6

Storage Management

LEARNING TREE INTERNATIONAL

Objectives

- Optimize memory management
- Store and retrieve objects using serialization



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

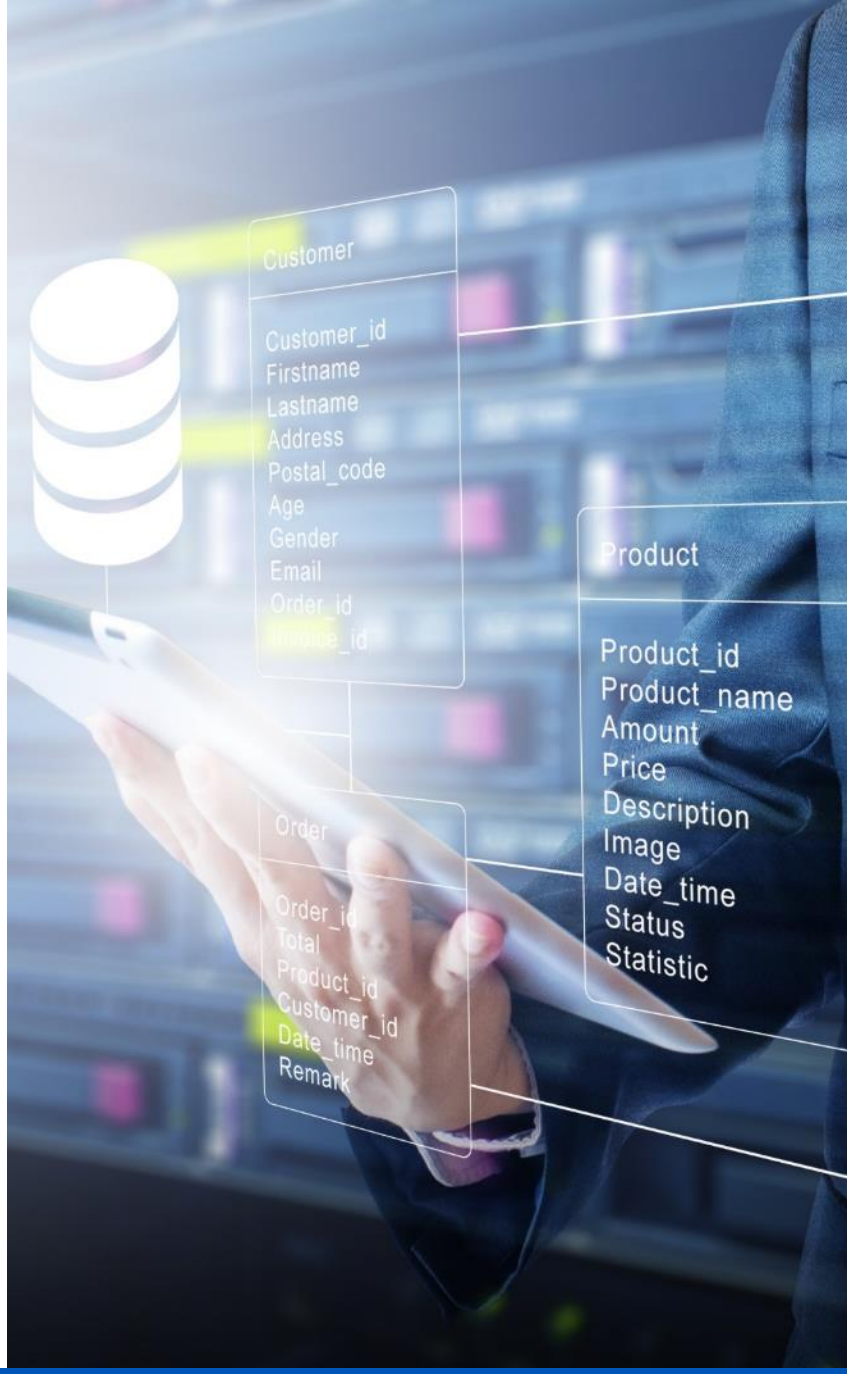
Contents

Memory Management

Smart Pointers

- Standard Library Smart Pointers
- R-value References and Move Semantics
- The Factory Pattern
- Overriding `new` and `delete`

- **Memory Layout and Derivation**
- **Serialization**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Smart Pointers

- **A *smart pointer* is an object that behaves like a pointer**
  - Allows programmer to take control of memory usage; e.g., to defer construction of an object until the pointer is traversed
  - Usually written as a template
- **A normal pointer uses operators like *, ->, ++, etc.**
- **A smart pointer class overrides these operators to take control of pointer behavior**
  - Usually written as a template to allow type-safe instantiation for different object types
  - A smart pointer will contain a real pointer plus any other information it requires
- **Have we already seen a smart pointer in the course?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

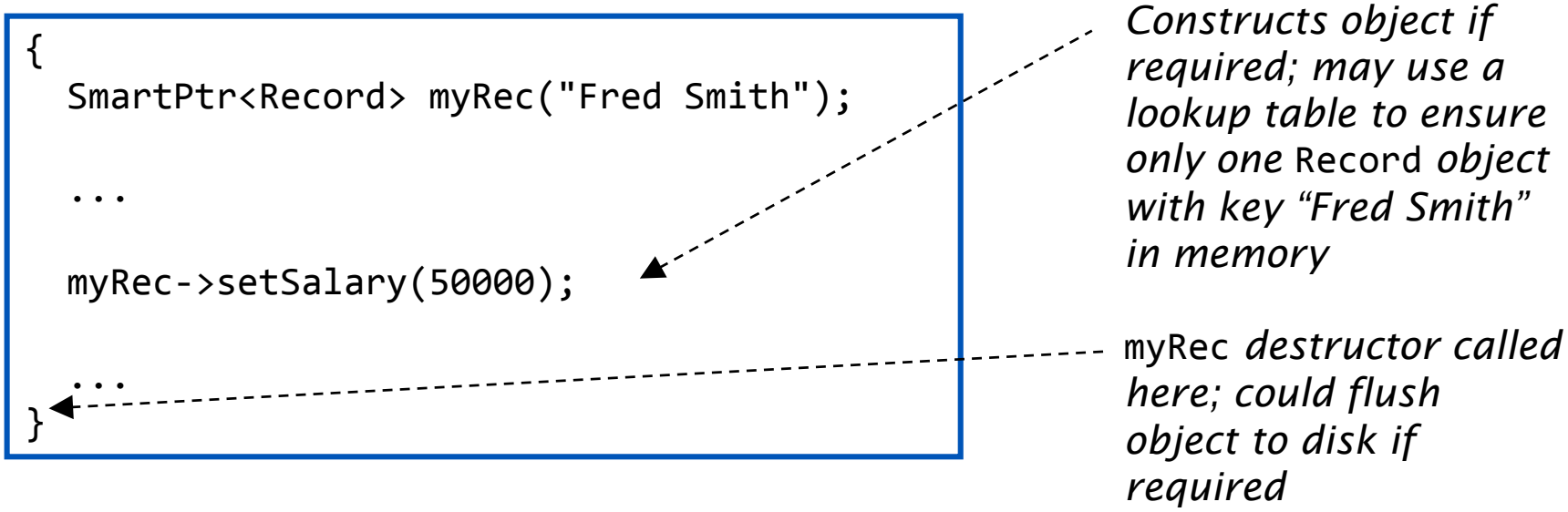
Smart Pointers

```
template <class T>
class SmartPtr {
public:
  T* operator->(){
    // smart pointer processing
  }
  T& operator*(){
    // smart pointer processing
  }
  ...
private:
  T* target;
  ...
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Smart Pointer: Example

- **A smart pointer can be used to allow access to objects whose data is stored on disk**
  - The smart pointer creates the object in memory only when required
  - The user should be able to use the smart pointer without having to know its smartness; i.e., using it as if it was pointing directly to the object



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

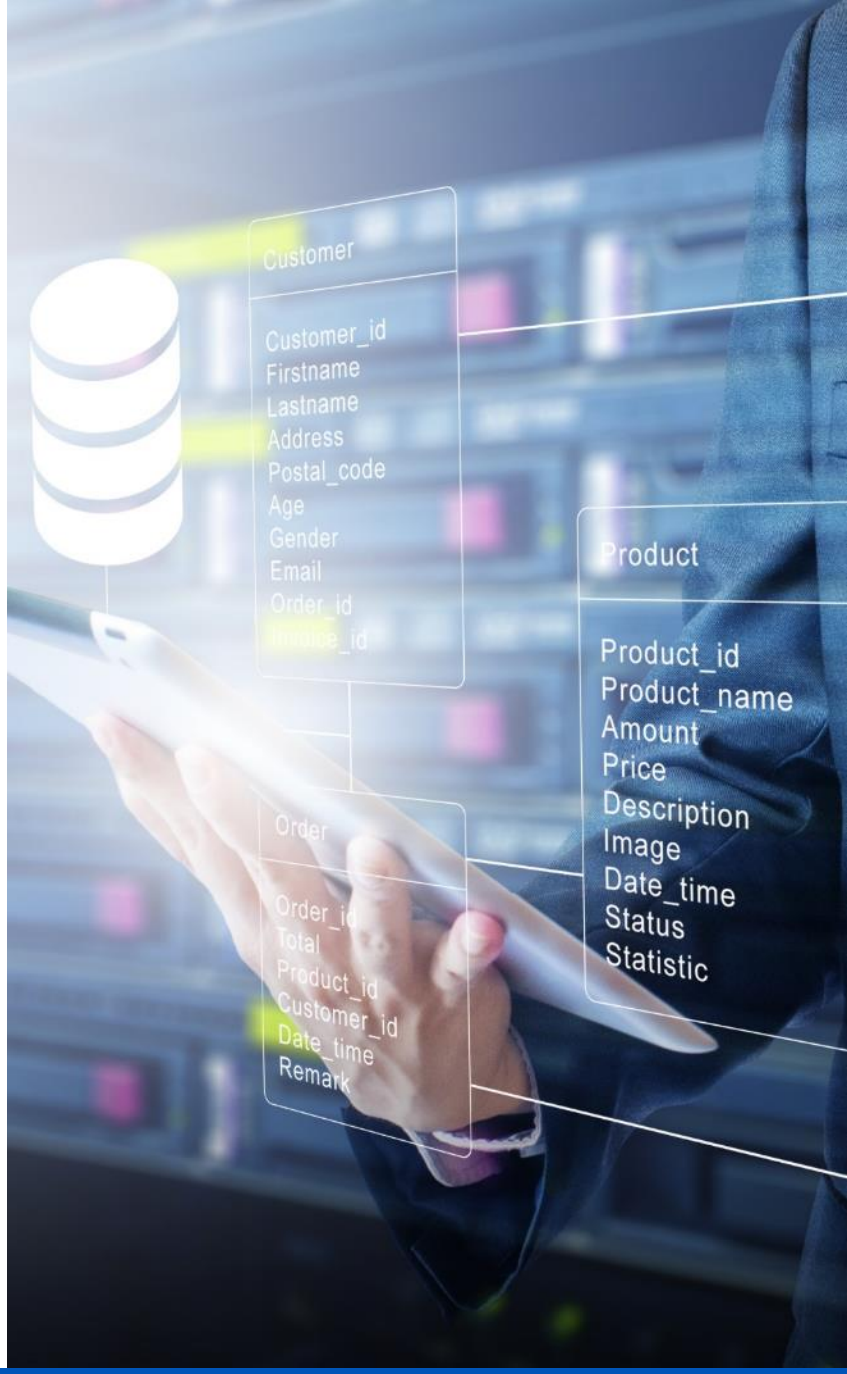
Memory Management

- Smart Pointers

Standard Library Smart Pointers

- R-value References and Move Semantics
- The Factory Pattern
- Overriding `new` and `delete`

- **Memory Layout and Derivation**
- **Serialization**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: The `unique_ptr` Template

- **The Standard Library provides the template class `unique_ptr`**
  - A smart pointer that destroys its target object when the `unique_ptr` goes out of scope
  - Provided in header `<memory>`
  - The destructor will be called whatever happens

```
#include <memory>
...
void f(){
  unique_ptr<X> myXPtr(new X);
  ...
  myXPtr->someFunc();
  ...

}
```

Initialize with a pointer

Use as normal pointer

Target X object's destructor called here; memory for the X object released

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Uses of `unique_ptr`

- **Simplifies memory allocation/deallocation**
  - No need to call `delete`
- **Allows dynamic objects to be deallocated after an exception has been thrown**
  - If the `unique_ptr` itself is a local variable, its destructor is called during stack unwinding
- **Should generally be used in preference to built-in pointers**



How about: *"I never really understood pointers, * and &. Now I can never go wrong with these smart ones"*

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Ownership of Target Object

- **A `unique_ptr` "owns" its target object**
  - So it is responsible for its destruction
- **Ownership can be transferred by moving the target of the `unique_ptr`**
  - Call `release()` to give up ownership
  - Call `reset()` to take on ownership

```
void f(){
  unique_ptr<X> myXPtr1(new X);

  unique_ptr<X> myXPtr2(myXPtr1.release());

  unique_ptr<X> myXPtr3;
  myXptr3.reset(myXPtr2.release());
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: shared_ptr

- **shared_ptr is a reference-counted smart pointer**
- **Can be assigned and copy-constructed**
  - They can safely be stored in containers, rather than raw pointers
- **Keeps track of references; destroys target object when last reference is destroyed**

```
void f(){
  shared_ptr<X> myXPtr1(new X);

  shared_ptr<X> myXPtr2(myXPtr1); // 2 references

  shared_ptr<X> myXPtr3;
  myXptr3 = myXPtr2;              // 3 references

  myXptr3 = nullptr;              // 2 references
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: `unique_ptr` and `shared_ptr`

- **Both `shared_ptr` and `unique_ptr` can be used for single objects or arrays**
  - `shared_ptr<Thing> ptrToSingleThing;`
  - `unique_ptr<Thing[]> ptrToThingArray;`
- **Can create a collection of `shared_ptr` and `unique_ptr`; e.g., in Standard Library containers**
  - `vector<shared_ptr<Thing>> thingPtrVector;`
  - The `shared_ptr` will be copied to/from the `vector`
  - The reference counting will, of course, be in action
  - Since you cannot copy a `unique_ptr`, it has to be *moved*
- **The reference counting of `shared_ptr` is thread safe**
  - If shared by multiple threads, only one will call its destructor



This *does not* make access to the referred object thread safe by default

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: make_unique and make_shared

- To avoid handling raw pointers even at creation time, two helper template functions are available

```
std::shared_ptr<Thing> tp(new Thing("paper", 32));
                              // raw pointer handling


auto tps(std::make_shared<Thing>("wood", 48));

auto tpu(std::make_unique<Thing>("iron", 67));
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

New in C++11: `weak_ptr`

- **The `weak_ptr` is used as a smart pointer in the same way as `shared_ptr`**
  - But does not own the object or have the responsibility of destroying it
  - And no guarantee that the pointed-to object still exists
  - To use the object, the `weak_ptr` must be temporarily converted to a `shared_ptr` with function `lock()`
- **Its main use is to break up cycles of `shared_ptr`**
  - If two objects point to each other with `shared_ptr`, their counters will never go down to 0 and the objects will never be destroyed
  - Replace the non-owning one with a `weak_ptr`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Old, Deprecated (Not-So-) Smart Pointer

- **The previous C++ standard contained the smart pointer template `auto_ptr`**



Do not use!

  - It is too fragile and only works for very simple situations
  - It is deprecated by C++11

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

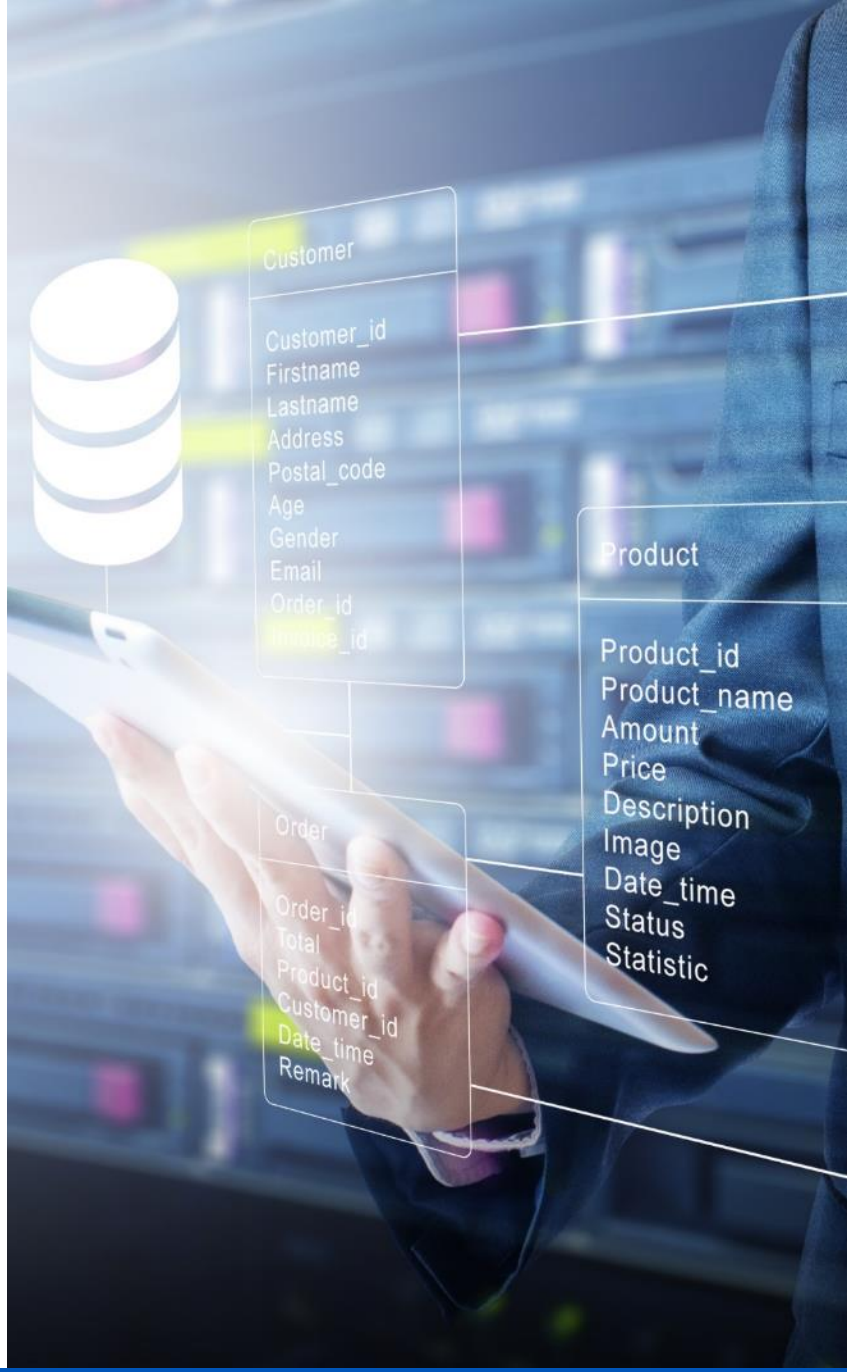
Memory Management

- Smart Pointers
- Standard Library Smart Pointers

R-value References and Move Semantics

- The Factory Pattern
- Overriding `new` and `delete`

- **Memory Layout and Derivation**
- **Serialization**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

R-Value References and Move Semantics

- **This type of reference is new in C++11 and is used to refer to a temporary object that can be modified, because it is about to go away and will not be used again**
  - An example to explain: Write a function that returns the sum of two **`Big`** objects
  - Note that the return type must be **`Big`** (i.e., not a reference)—we cannot return a reference to **`b`** since it is going away when `sum` terminates
  - The copying of the return value from **`b`** used to be done by the copy constructor
  - In C++11, the *move constructor* will be used instead, by default
  - This can avoid unnecessary copying of data

```
Big sum(const Big& b1, const Big& b2){
        Big b = //calculate the sum of b1 and b2
        return b;
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Move Constructor

- **The move constructor takes an r-value reference as its parameter**
  - The assumption is that any parameter passed by r-value reference to a method will not be used again after the method exits; it can therefore be modified by the method

```
class Big {
   BigData *ptr;
   int size;
   ...
   public:
   Big (Big&& rvr) : ptr(rvr.ptr), size(rvr.size) {
      rvr.ptr = nullptr;
      rvr.size = 0;
      ...
   }
   ...
};
```

Move the data from `rvr` to `*this` by copying pointer, not data. This is OK, since the data of `rvr` is not going to be used anymore.

We should also implement the move assignment **`operator=`**

- **Why are we setting `rvr.ptr` to `GreaterComparableptr` and `rvr.size` to `0`?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

R-Value and Move Semantics: Best Practices

- **If a class is significantly big and stores data that can be moved; e.g., stored via pointer**
  - Define your own move constructor and move assignment

```
Big::Big(Big&& r);              // move constructor
Big& Big::operator=(Big&& r); // move assignment
```

  - Both will then automatically be used where appropriate: return values, moving data between containers, and other library features that use `move`
  - Make sure that the parameter object **`r`** above is modified to be empty
    - It needs to be in a consistent state, even when emptied
    - For example, the destruction of it should not affect any of the moved data

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

R-Value and Move Semantics: Best Practices

- **If you want to move an object yourself, there is a template library function `move` that will create the r-value reference**
  - Be careful how you use the object that you have moved from

```
void slowswap(Big& b1, Big& b2) {      // three Big copies
      Big tmp = b1; // copy constructor
      b1 = b2;        // copy assignment
      b2 = tmp;     // copy assignment
}

void fastswap(Big& b1, Big& b2) {          // no Big copies!
      Big tmp(move(b1)); // move constructor
      b1 = move(b2);     // move assignment
      b2 = move(tmp);    // move assignment
}
```

  - There is a `std::swap` library template function that does the job of `fastswap` above, so there is no need to write this particular one, but it is a perfect example of the benefits of move semantics

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Aside: `nullptr` vs. `std::optional` (C++17)

- **It might be a bad idea to return `nullptr` from a function to indicate non-existence of a result**
  - The inventor of the null pointer, Tony Hoare, says: "*I call it my billion-dollar mistake*"
  - C++17 introduces a template for *optional* values
  - *Forces* caller to think about non-existence
  - No risk of dereferencing a null pointer (result undefined)

```
std::optional<int> getSize();

//…

auto result = getSize();

if (result.hasValue()) {
    int size = result.value(); // would throw exception if no value present
    //…
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

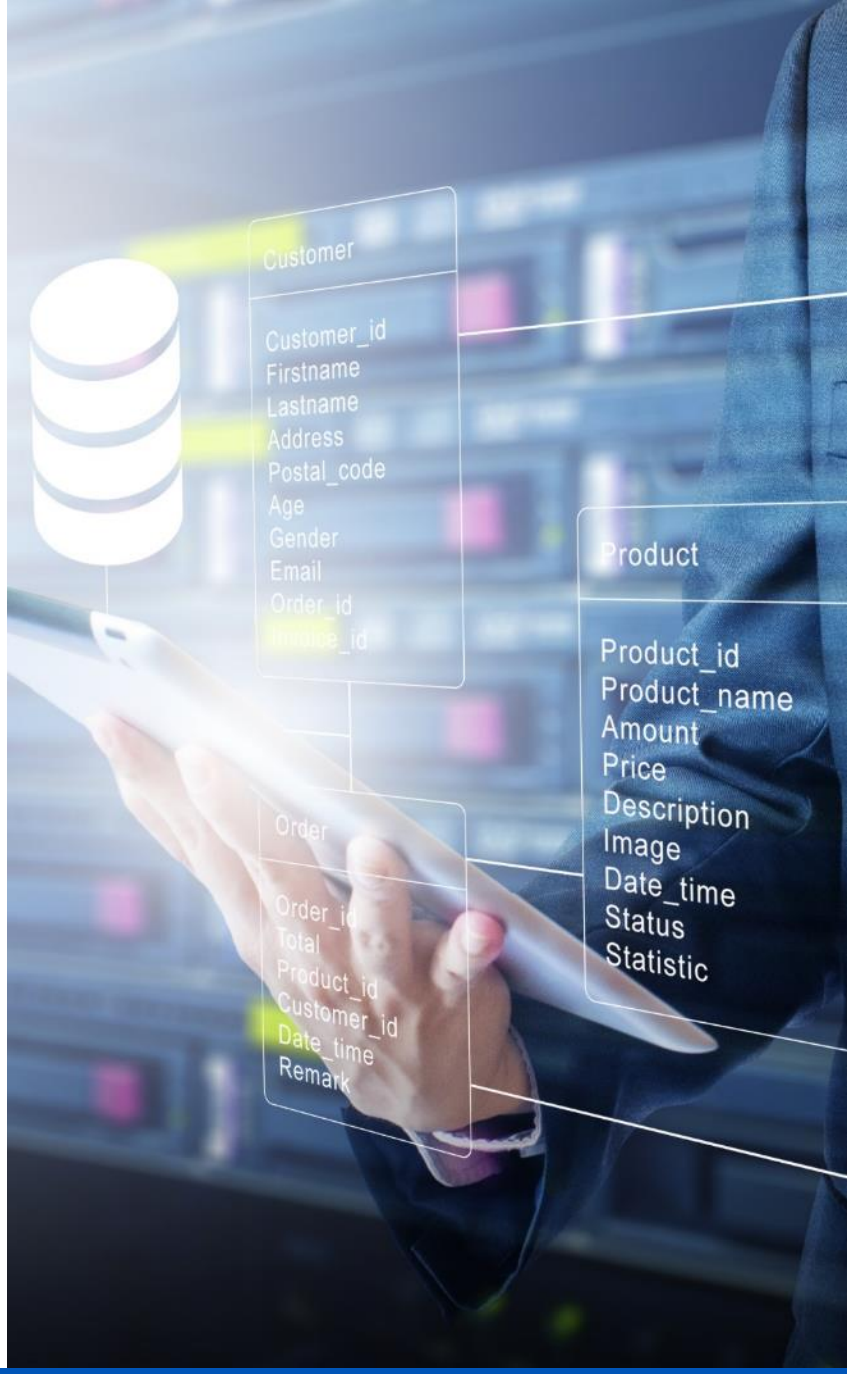
Memory Management

- Smart Pointers
- Standard Library Smart Pointers
- R-value References and Move Semantics

The Factory Pattern

- Overriding `new` and `delete`

- **Memory Layout and Derivation**
- **Serialization**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Pattern: Factory

- **Sometimes, it is a good idea *not* to use `new`, but to delegate the creation to a method, the *Factory pattern***
  - We might not (want or need to) know exactly what kind of object we want
  - We might not know whether we can reuse a recyclable/sharable object
  - We might not know whether we can share an object with others

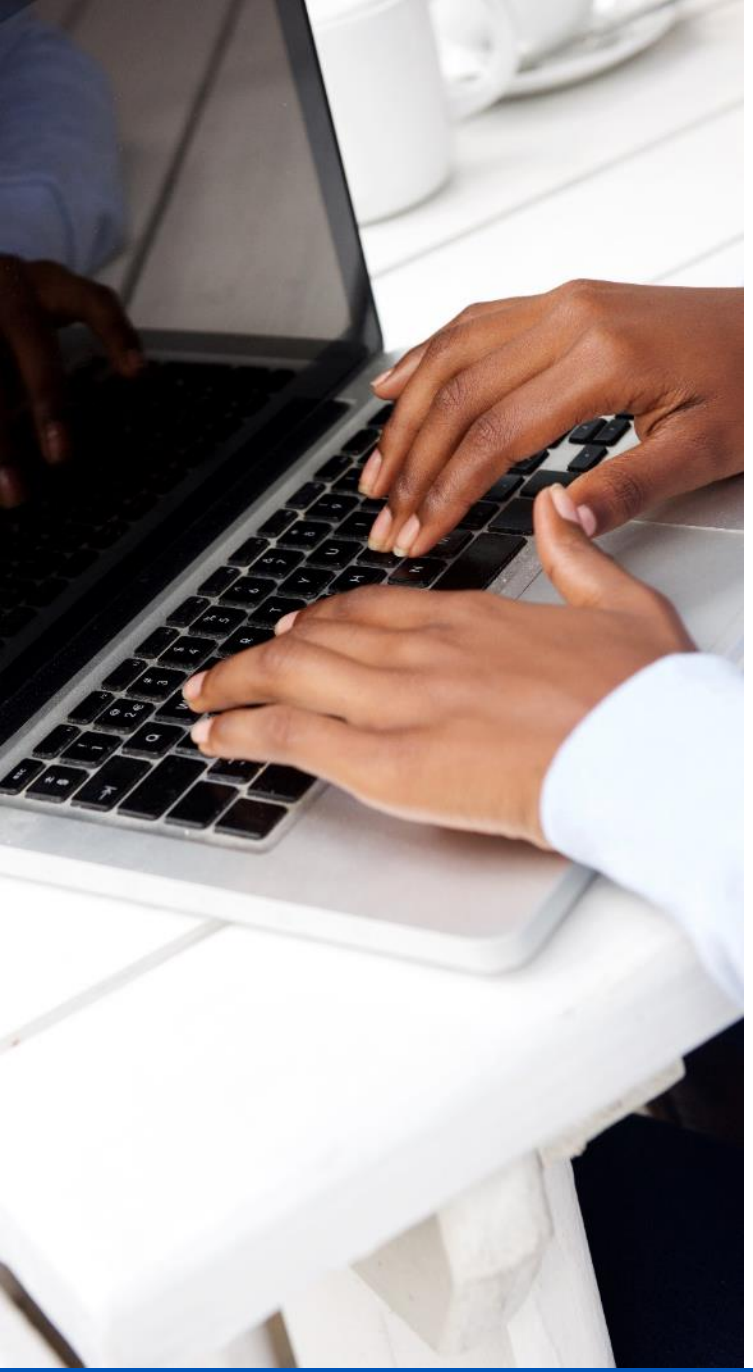
```
auto iter = mycontainer.begin();
                // creates the right kind of iterator
                // depending on what myContainer needs

Greeting *g = GreetingFactory.create();
                // create a greeting appropriate to time-of-day

Connection *c = ConnectionManager.connect("DB-id456");
                // maybe share or reuse?

Person *p = PersonFactory.makePerson(45);
                // subclass of Person according to age parameter
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Hands-On Exercise 6.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 6.1: Memory Management

In this exercise, you will modify a console application

- The application has memory leaks caused by the use of pointers
- A pointer to memory is inadvertently lost when an error occurs, which leads to a memory leak

You will

- Build and run the program, identifying the source of the leak
- Fix the leak, using the `shared_ptr` template
- Further modify the program, using a form of the *Factory Method* pattern to delegate the creation of the correct type of object

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

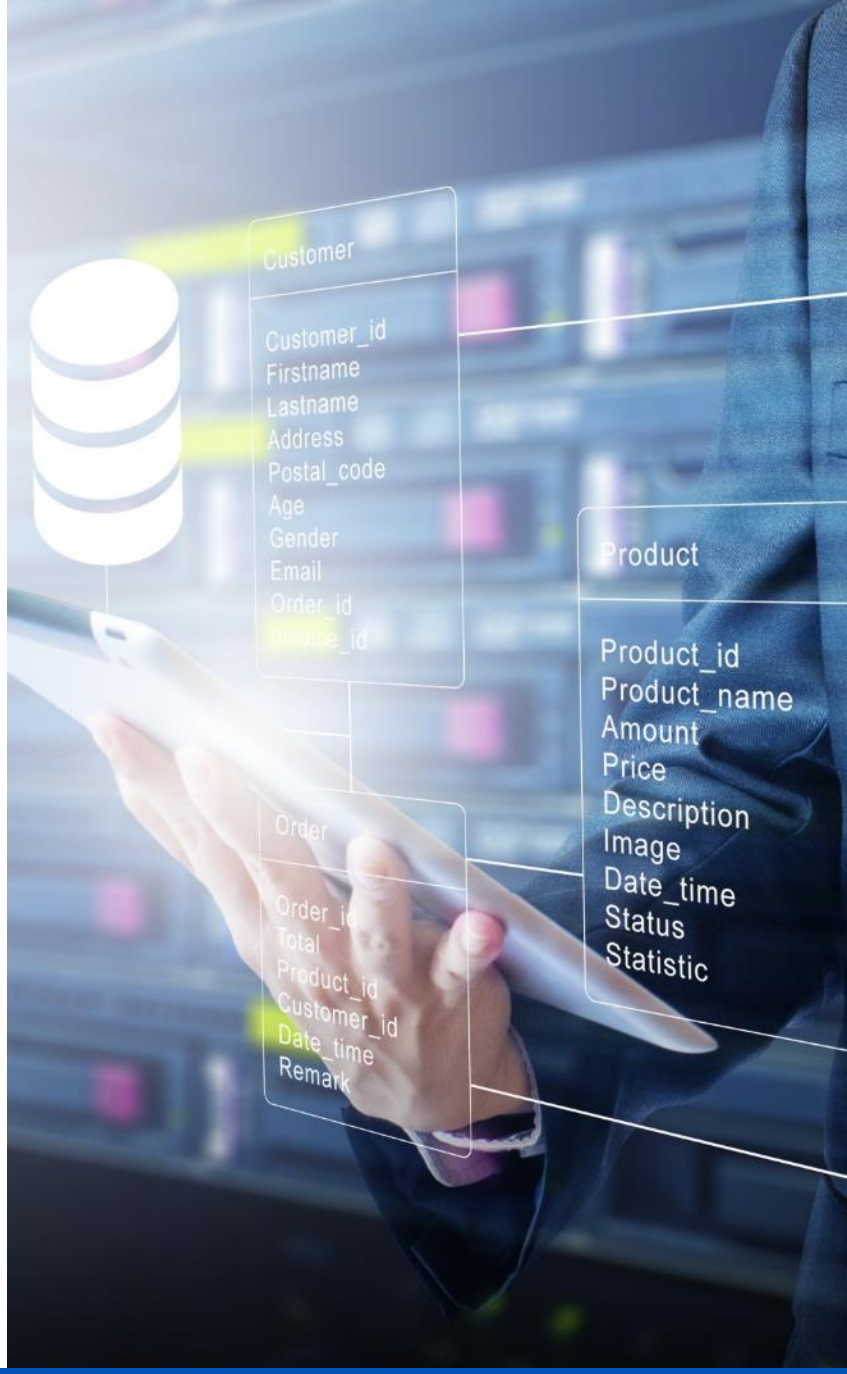
Contents

Memory Management

- Smart Pointers
- Standard Library Smart Pointers
- R-value References and Move Semantics
- The Factory Pattern

Overriding `new` and `delete`

- **Memory Layout and Derivation**
- **Serialization**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

new

- **Objects are created dynamically using a `new` expression**
  - `MyClass* myPtr = new MyClass;`
- **The `new` expression**
  - Invokes `operator new`, which returns the address at which the object will be built
  - Invokes a constructor to initialize the object
- **The default `operator new` returns an address from heap storage**
  - The programmer can redefine the default version of this operator to take control of the memory allocation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Redefining new

- **The default `operator new` has the following signature**
  - `void* operator new(size_t n);`
    - Where `n` is the number of bytes to be allocated for the object
    - The `new` expression supplies this value
- **This operator can be redefined for a specific class**
  - For example, you may want objects of a specific class to be allocated from shared memory



If you override **`new`** in this way, you must also override **`delete`**

```
class MyClass{

  ...
  void* operator new(size_t n){
    void* address;
    address = ... // an appropriate free address
    return address;
  }
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Overloading new

- **Other information can be passed to your `operator new`**
  - For example, you might want to pass as an argument the address at which the object is to be allocated
  - This is done by adding an extra parameter to `operator new`
  - Overloading allows any type and number of parameters

```
class MyClass {
  ...
  void* operator new(size_t n, void* address){
    ...
    return address;
  }
};
```

- **This overloading is invoked by a *placement* `new` *expression***
  - The constructor is run as usual

```
void* buf = reinterpret_cast<void*>(0xFF000030);
MyClass* ptr = new (buf) MyClass;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Optional Details

- **Appendix B contains a more detailed overview of other aspects of overloading `new` and `delete`**
  - It is quite rare that a program requires its own memory allocation

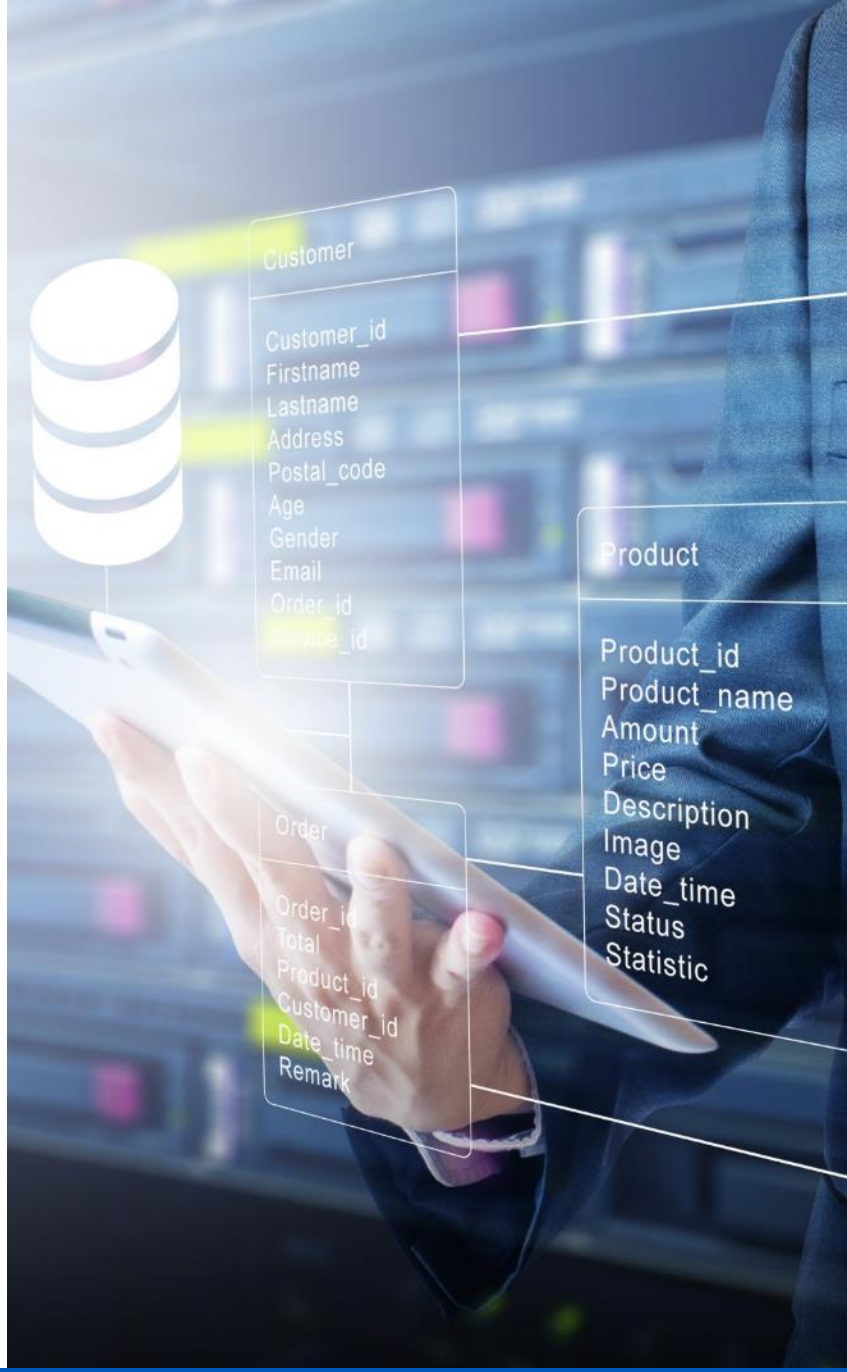
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Memory Management

Memory Layout and Derivation

- Serialization



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Compiler Dependence

- **The C++ standard does not specify how a compiler should lay out objects or classes in memory**
  - It does specify behavior that must emerge from the compiler's strategy
  - This leads to certain approaches being favored

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

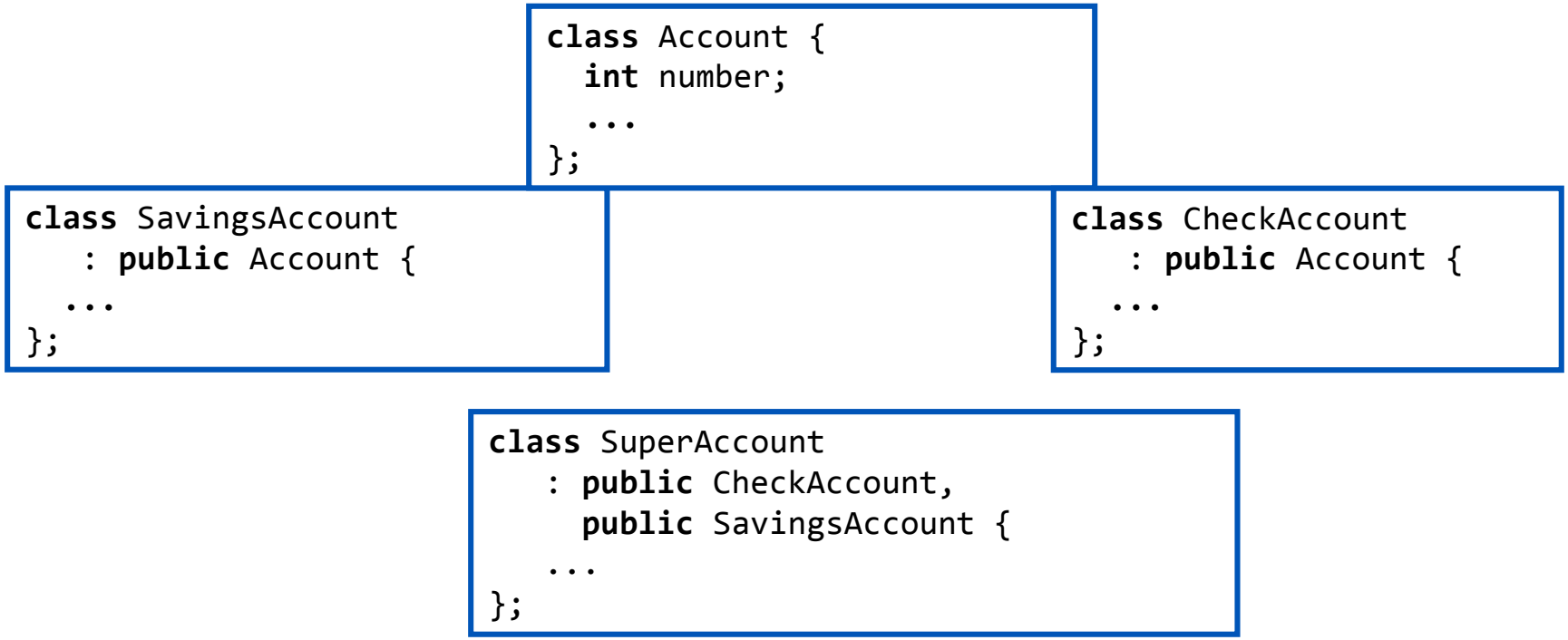
Memory Layout and Virtual Functions

- **Objects of a class without any virtual functions need only contain its data members**
  - These classes are sometimes called *lightweight* and, for compatibility reasons, are laid out the same way as a corresponding C `struct`
  - This kind of class is easy to optimize in many ways, and is very often used in utility libraries
    - Note that both `std::string` and STL use neither inheritance nor virtual functions; template instantiation is done at compile time
- **When a first virtual function is added to a class, each object will need a *virtual pointer*, pointing to a *virtual table***
  - The exact mechanism is implementation dependent
  - There is a single virtual table for each class involved
- **The virtual table contains the necessary information to make polymorphism, `dynamic_cast`, and runtime type information (RTTI) work**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Implementation Inheritance

- **Multiple implementation inheritance means the capability of inheriting from many classes that are not pure interface classes**
  - Java and C# have explicitly forbidden this
  - It introduces some complex situations, particularly when the base classes have common ancestors
- **How many `number` members are there in a `SuperAccount`?**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Virtual Base Classes

- **Virtual base classes can fix the problem with multiple common ancestors**
  - But there is almost always a reasonable way to avoid using multiple implementation inheritance
  - Note that Java and C# both allow multiple *interface* inheritance
- **How could we redesign to avoid multiple inheritance in the `Account` hierarchy?**

```
class Account {
  int number;
  ...
};
```

```
class SavingsAccount
   : public virtual Account{
  ...
};
```

```
class CheckAccount
   : public virtual Account{
  ...
};
```

```
class SuperAccount
   : public CheckAccount,
     public SavingsAccount {
   ...
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More on Memory Layout and Inheritance

- **Appendix B contains a full illustrated story of**
  - Memory layout
  - Virtual pointers and tables
  - Single inheritance
  - Multiple inheritance

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

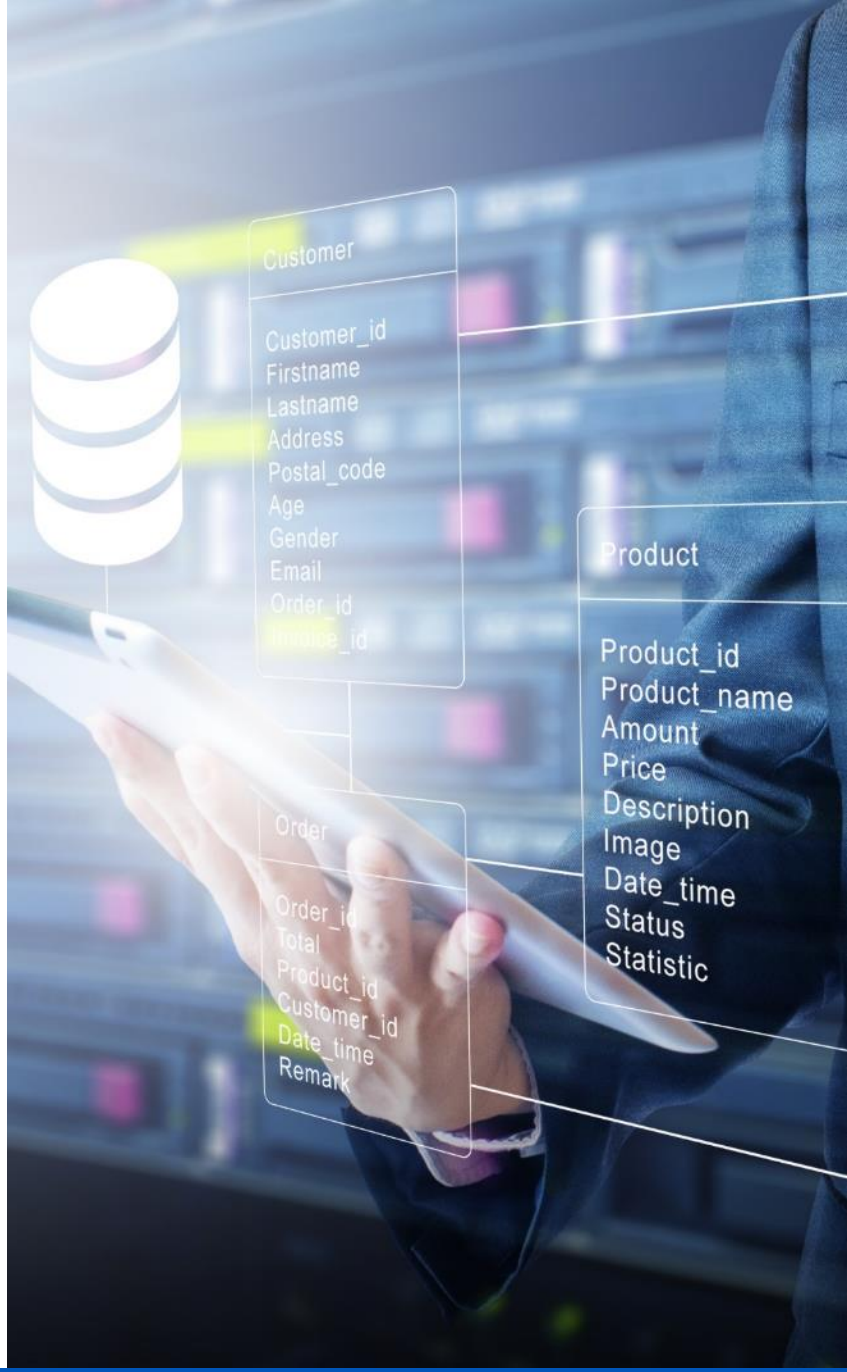
Contents

- Memory Management
- Memory Layout and Derivation

Serialization

Introduction

- Designing a Serializable Class
- Serializing
- Deserializing



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

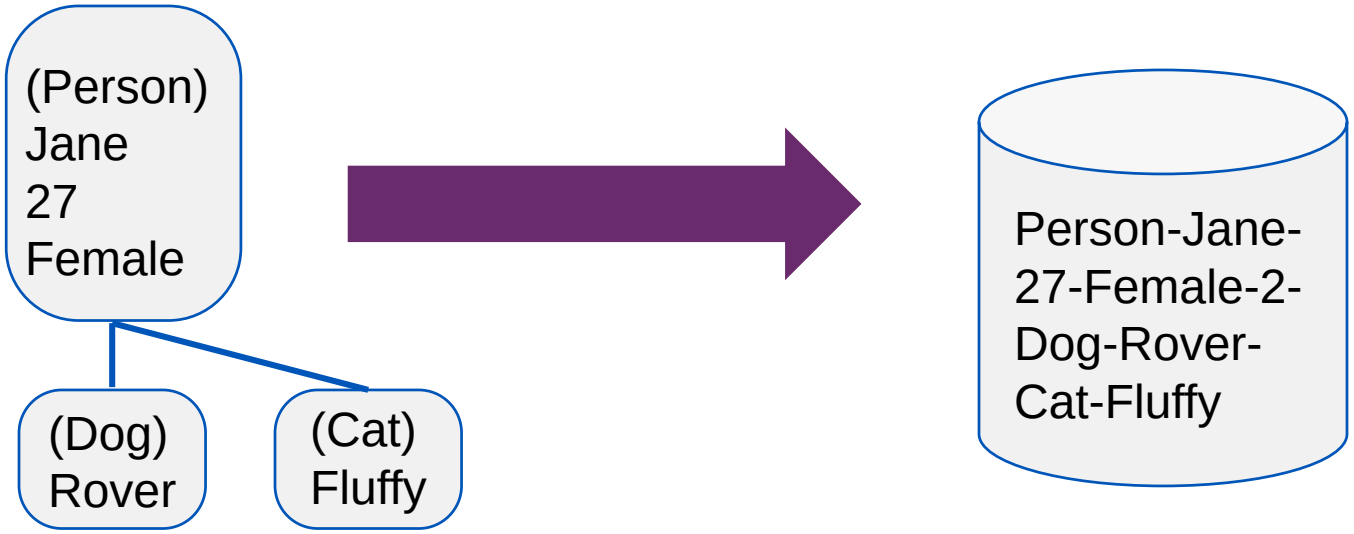
Serialization: Uses

- **Primarily used for storing large, complex, tightly coupled structures**
  - Intended for single-user access
  - Whole structure accessed at once
- **Instead of being stored, the serialized data can be sent and received over a network connection**
- **Examples**
  - Word-processing document
  - Spreadsheet
  - Graphics package diagram
  - Remote object access

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Serialization

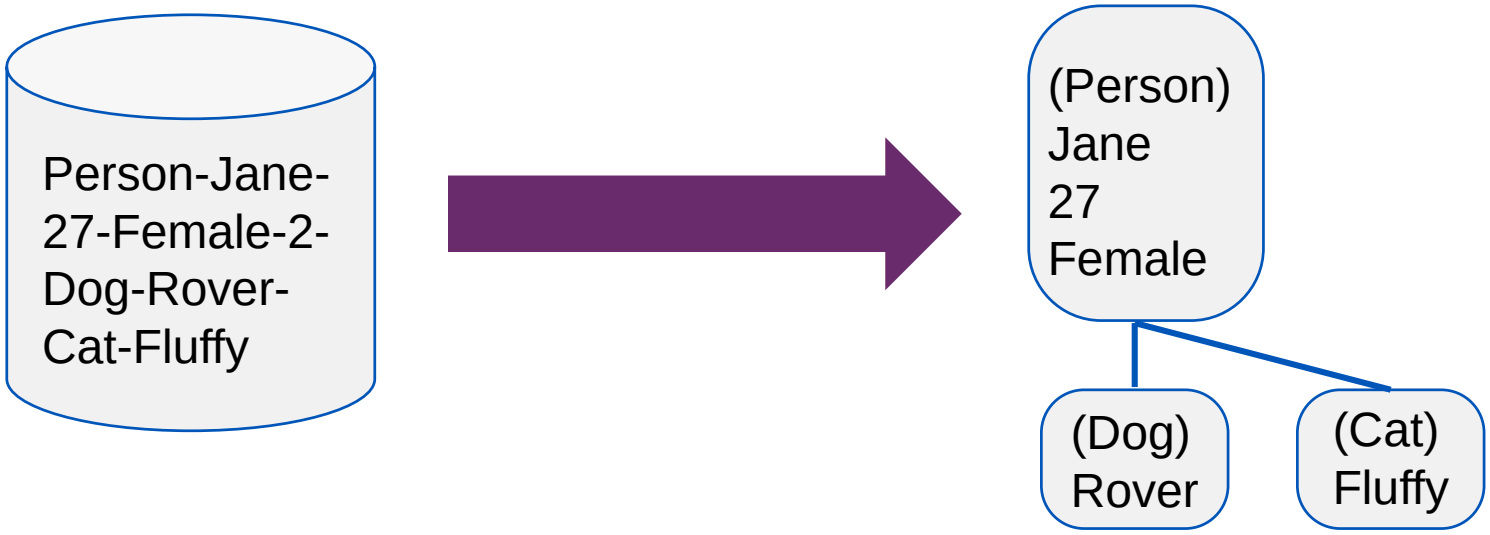
- **An object can be *serialized* to a file by storing**
  - Its class name
  - Each attribute or linked object in turn
    - Linked objects serialize themselves
    - A collection stores its multiplicity (number of members)
  - And so on recursively, until the entire structure is stored



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Deserialization

- **Deserialization is the opposite process**
  - Class names and collection multiplicities are necessary in order to re-create objects correctly



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Serialization: Implementation

- **Serialization libraries can be obtained from compiler vendors and third-party vendors; e.g.:**
  - Microsoft
  - Boost
- **This section describes how to implement serialization using the Boost classes**
  - The principles are the same for any implementation
  - They vary, however, in their intrusiveness; i.e., how much the code becomes dependent upon the serialization library being used
  - Boost is unusually non-intrusive
- **The Boost organization provides free, peer-reviewed, portable C++ open-source libraries; some of it is now part of C++11**
  - Boost covers many areas, not just serialization
    - Numerics
    - Date and time
    - Threading

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

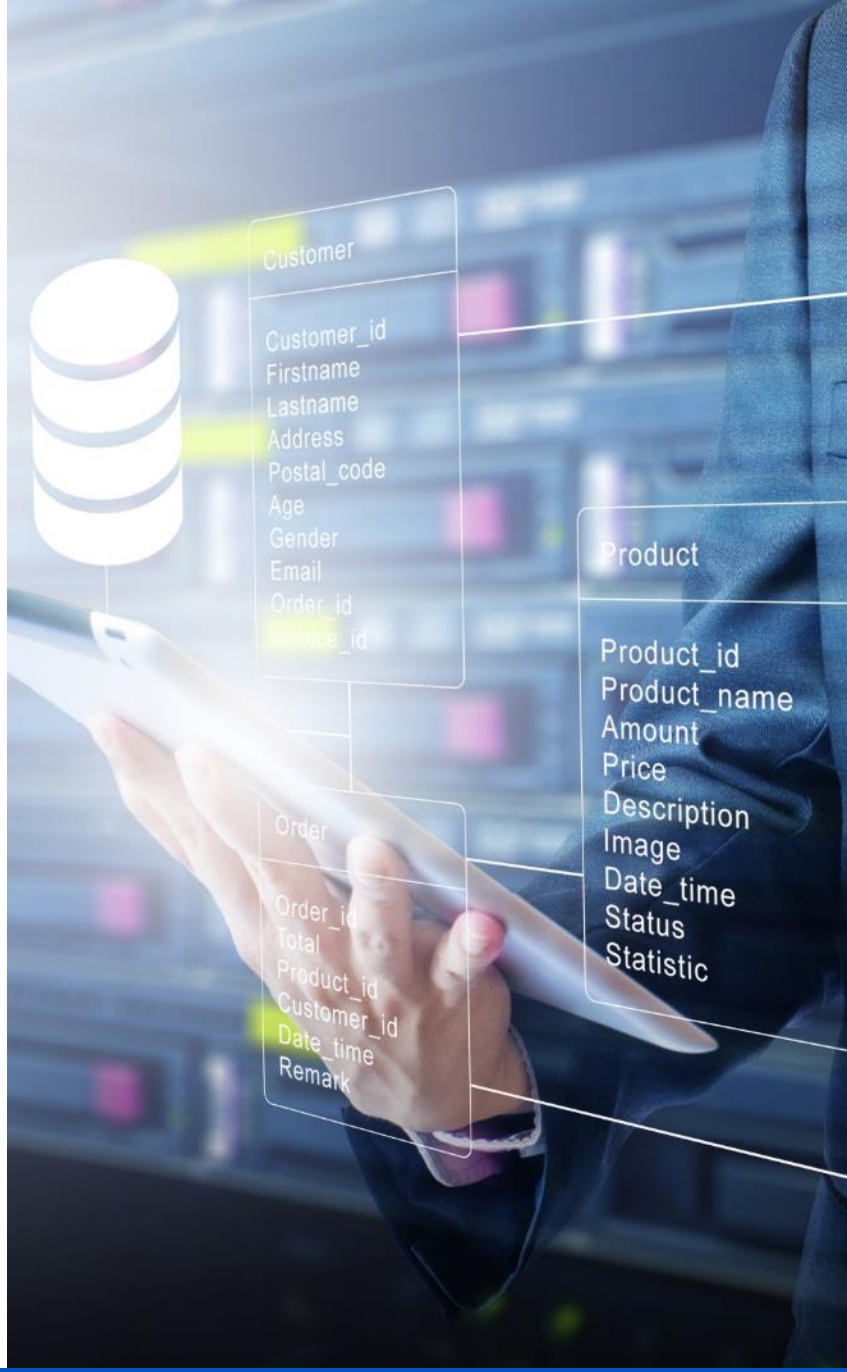
- Memory Management
- Memory Layout and Derivation

Serialization

- Introduction

Designing a Serializable Class

- Serializing
- Deserializing



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Designing a Serializable Class

- **Add a private, template function `serialize`**
  - Serializes each member of the class by calling `operator&`
  - Called recursively from the serialization archive classes
  - Can also be defined as a non-member template function in namespace `boost::serialization`
    - ```
      template <class Archive>
      void serialize(Archive& ar, const unsigned int ver);
      ```
    - Not necessary to change class definition to add serialization
    - STL collections are made serializable in this way
- **`operator&` is defined in the Boost serialization archive classes**
  - `boost::archive::text_oarchive`—textual output stream
  - `boost::archive::text_iarchive`—textual input stream
  - Can be invoked on built-in types, class objects, pointers, STL collections, and arrays; XML archive streams are also provided
- **Make class `boost::serialization::access` a friend of your class**

XML = extensible markup language

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.
      ```

A Serializable Class

```
#include <boost/serialization/list.hpp>
class MyClass {
public:
...
private:
  // data members of class
  int x;
  std::list<std::string> collection;
  MyOtherClass* ptr;
  ...
  template<class Archive>
  void serialize(Archive& ar,
                 const unsigned int version){
    ar & x;
    ar & collection;
    ar & ptr;
  }
  friend class boost::serialization::access;
};
```

Code for serializing STL lists

Add this function to enable serialization

Serialize data members

To allow call to the private `serialize` *method*

- **The serialization code can also be put outside of the class to allow even less intrusiveness**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

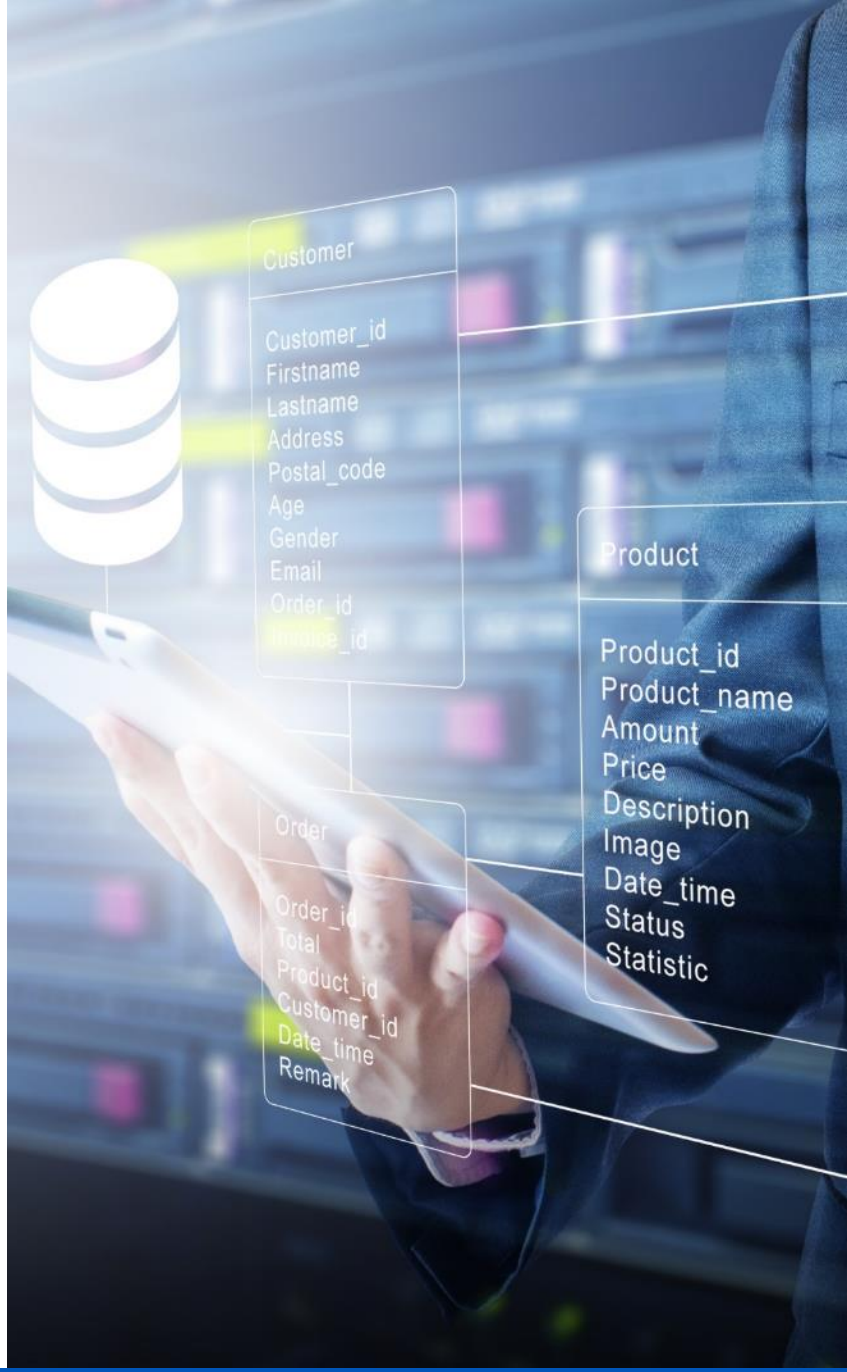
- **Memory Management**
- **Memory Layout and Derivation**

Serialization

- Introduction
- Designing a Serializable Class

Serializing

- Deserializing



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Serializing

- **Create a `boost::archive::text_oarchive`**
  - Initialized with an output file stream
- **Use `operator<<` to serialize one or more objects to the archive**
- **Close the archive**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Serializing: Example

```
class MyClass {
...
};
```

```
#include <fstream>
#include <boost/archive/text_oarchive.hpp>

void f(){
   MyClass myObject;
   ...
   ofstream ofs("file.dat");
   boost::archive::text_oarchive ar(ofs);

   ar << myObject;
   ...
} // archive closed as it goes out of scope
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

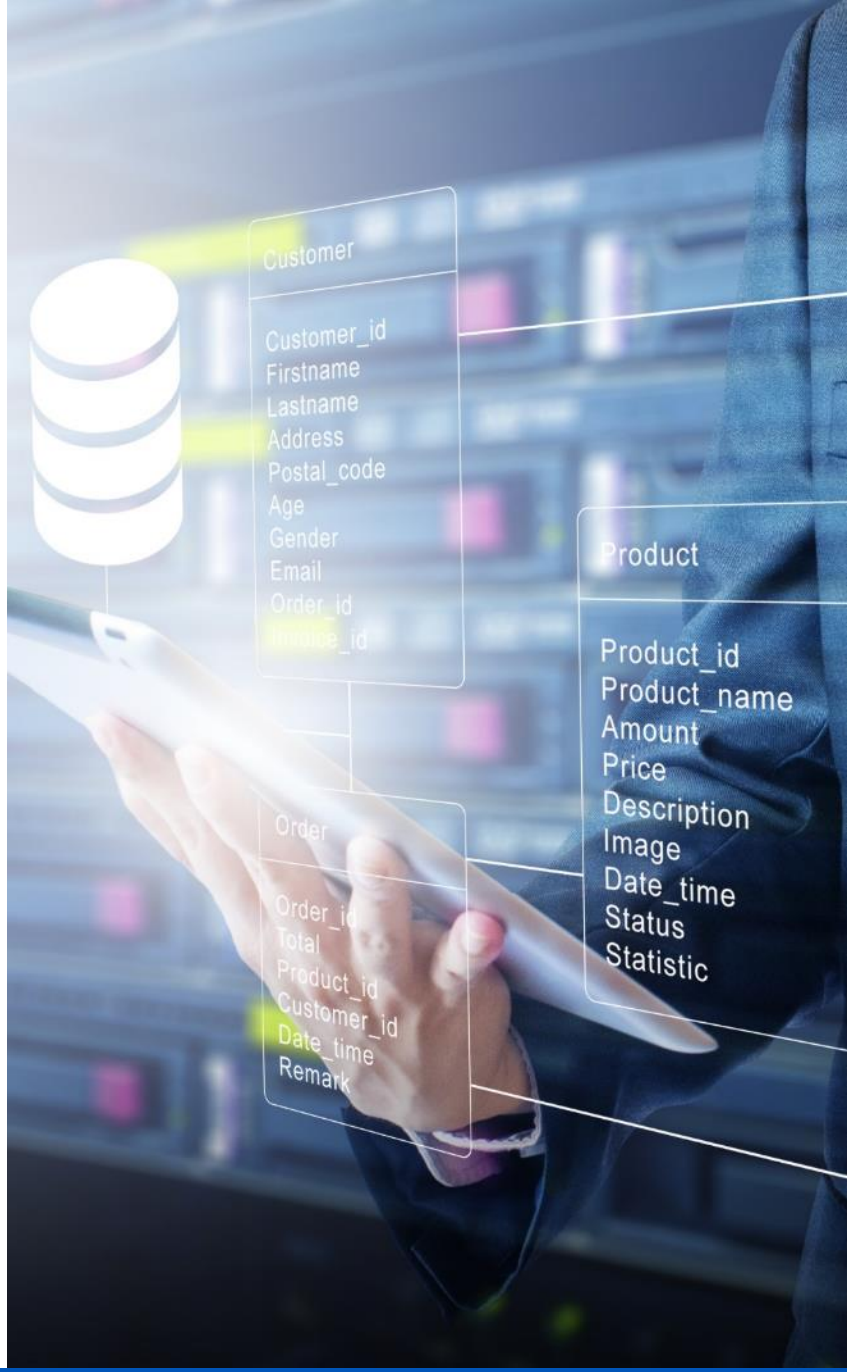
Contents

- Memory Management
- Memory Layout and Derivation

Serialization

- Introduction
- Designing a Serializable Class
- Serializing

Deserializing



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Deserializing

- **Create a `boost::archive::text_iarchive`**
  - Initialized with an input file stream
- **Use `operator>>` to deserialize one or more objects from the archive**
  - Must correspond to the type of objects that were serialized into the archive
- **Close the archive**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Deserializing: Example

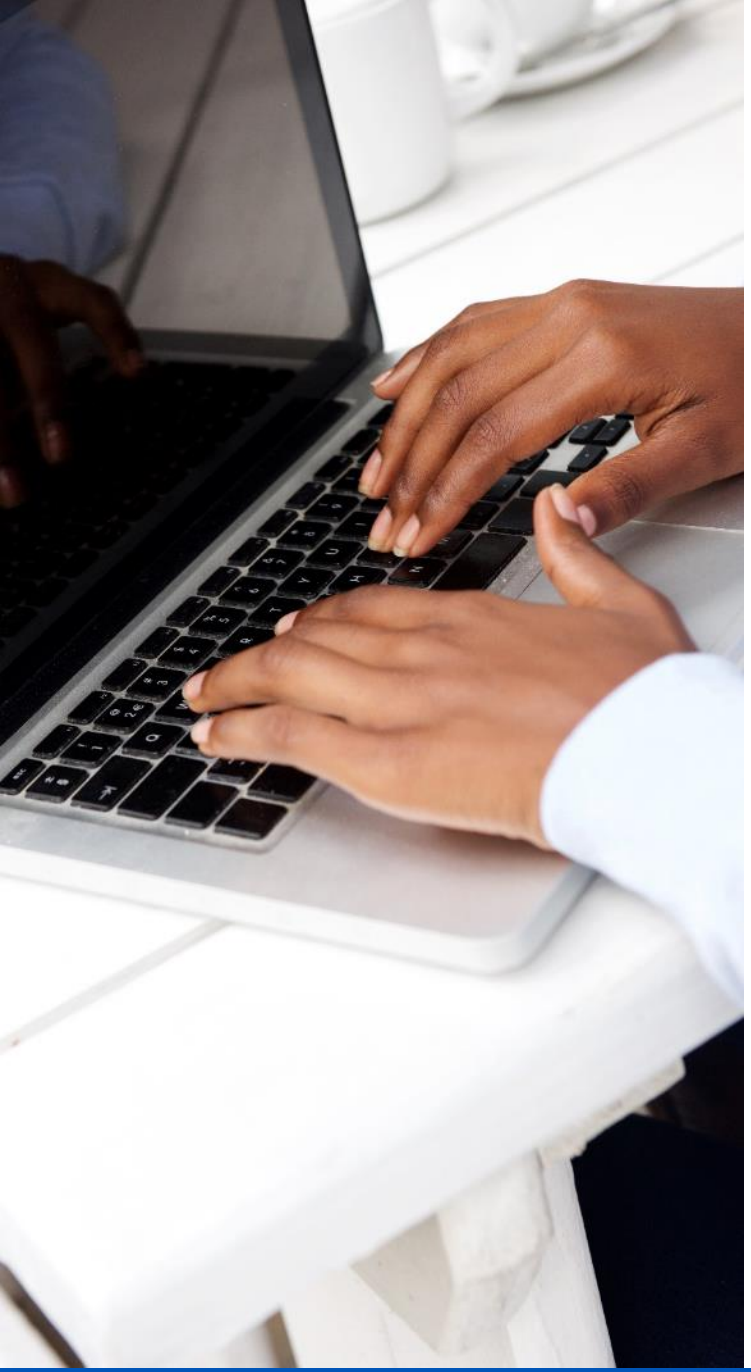
```
class MyClass {
...
};
```

```
#include <fstream>
#include <boost/archive/text_iarchive.hpp>

void f() {
   MyClass myObject;
   ...
   try{
      ifstream ifs("file.dat");
      boost::archive::text_iarchive ar(ifs);

      ar >> myObject;
   }
   catch(...){
      // possibly non-existent file
   }
   ...
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Optional Hands-On Exercise 6.2

Exercise Manual

In your Exercise Manual, please refer to Optional Hands-On Exercise 6.2: Serialization

In this exercise, you will be working on a console application

- The application creates, modifies, and displays some objects

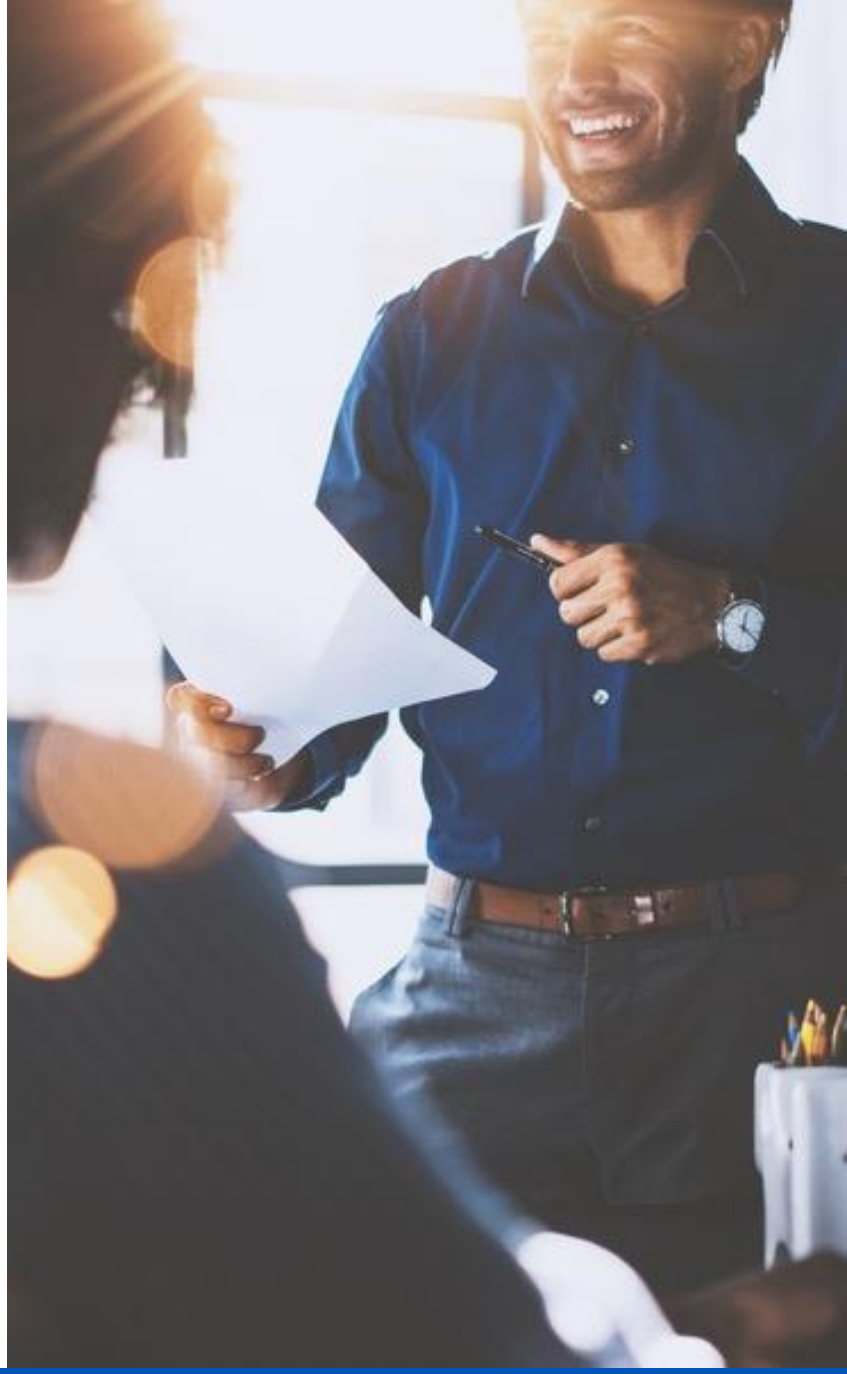
You will

- Build the application, run it, and familiarize yourself with it
- Modify it to store in an archive the objects it creates
- Modify it to retrieve objects from an archive, if one exists

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- Optimize memory management
- Store and retrieve objects using serialization



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Chapter 7

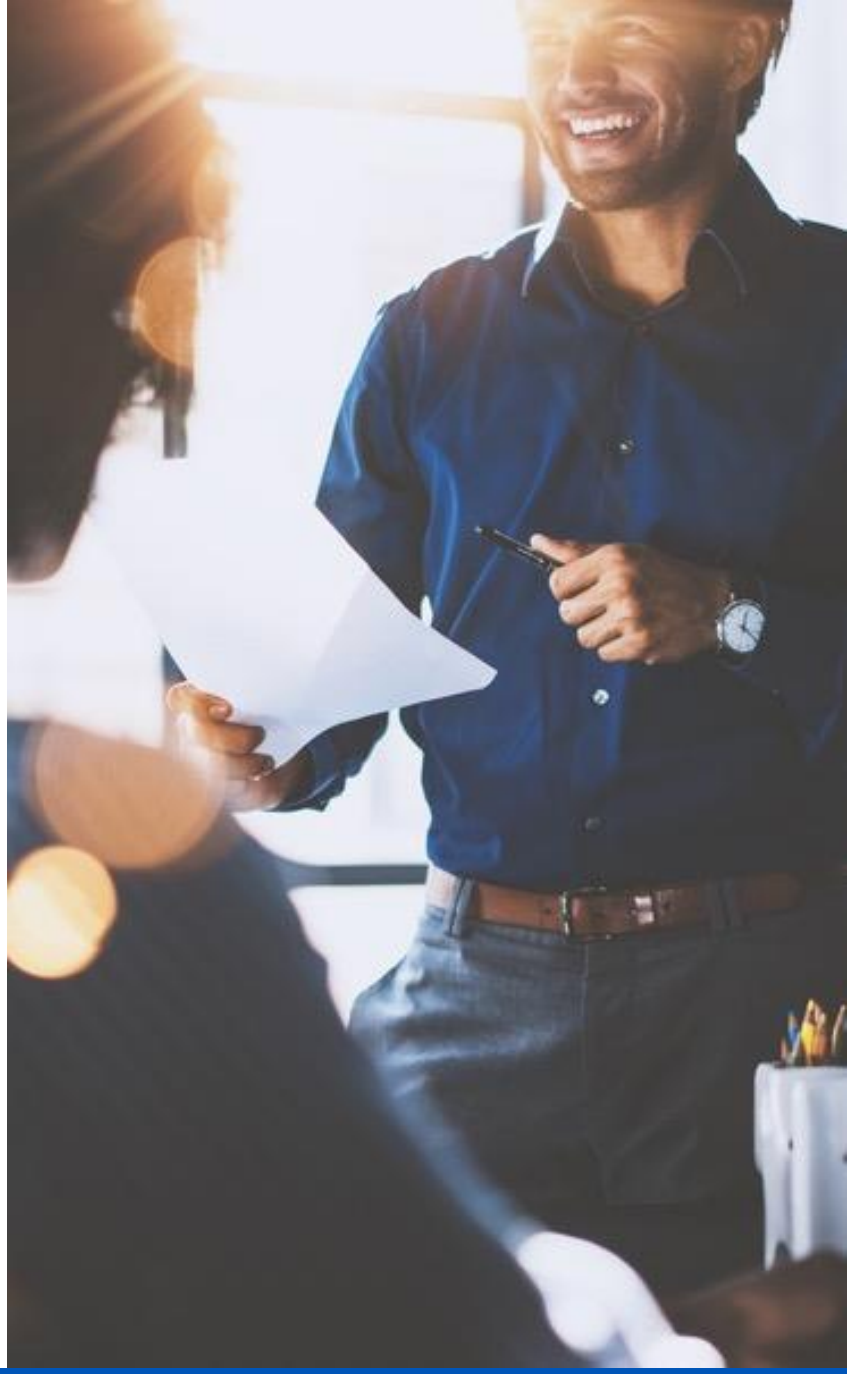
Patterns, Part 3: Composite and Visitor Patterns

LEARNING TREE INTERNATIONAL

Objectives

Learn about patterns that are often used together

- **The Composite pattern**
  - A pattern for simplifying the handling of recursive, whole-part structures
- **The Flyweight pattern**
  - A pattern that permits efficient handling of structures with a large number of objects with common state
- **The Visitor pattern**
  - How to separate operations from state in situations where they may need to change independently



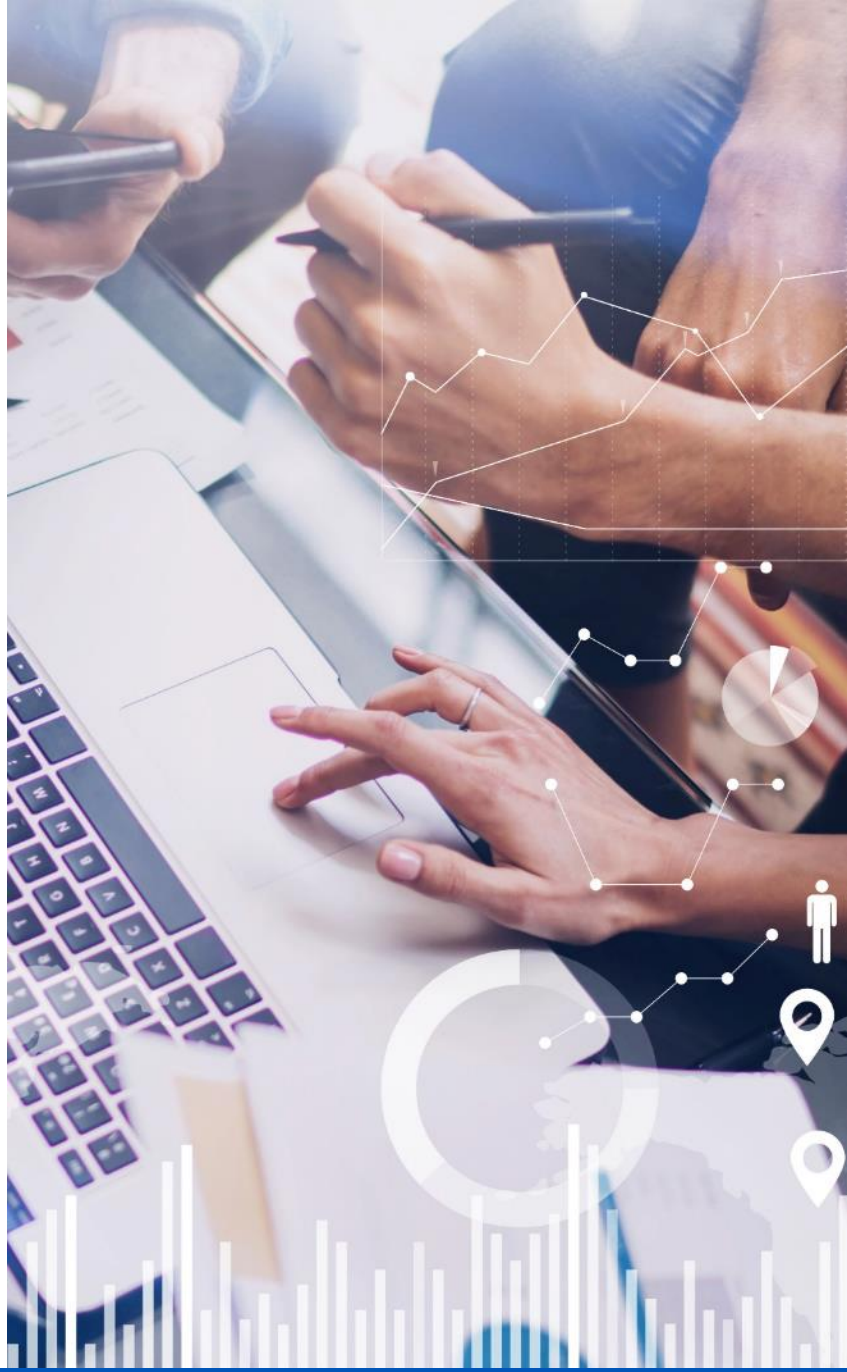
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Composite Pattern

- Identification
- Problem
- Solution
- Consequences

- **Visitor Pattern**



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

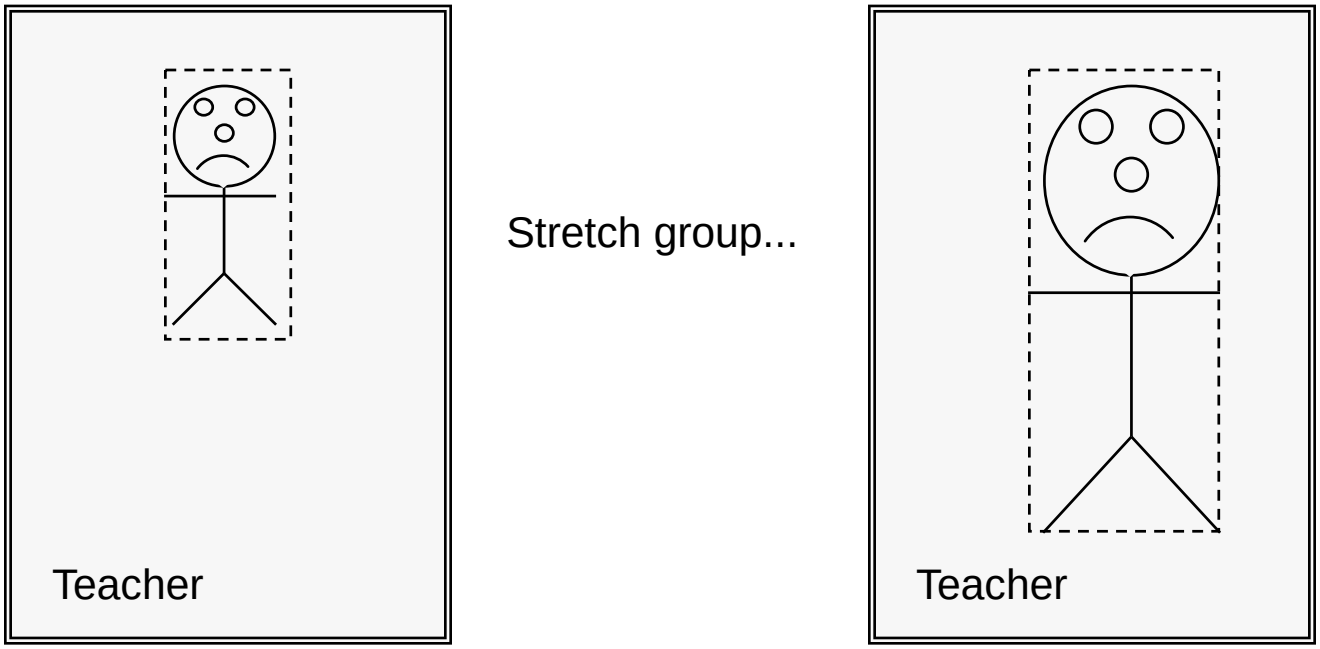
Identification

- **Name**
  - Composite
- **Classification**
  - Structural
  - Object
- **Intent**
  - Compose objects into a tree structure to represent a whole-part hierarchy
  - Allow clients to treat individual objects and compositions in the same way

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Motivation

- **A graphics application allows a user to form primitive elements such as lines and ellipses into groups**
  - A group can be manipulated in many of the same ways as primitive elements
    - Stretching, moving, hiding, deleting, etc.

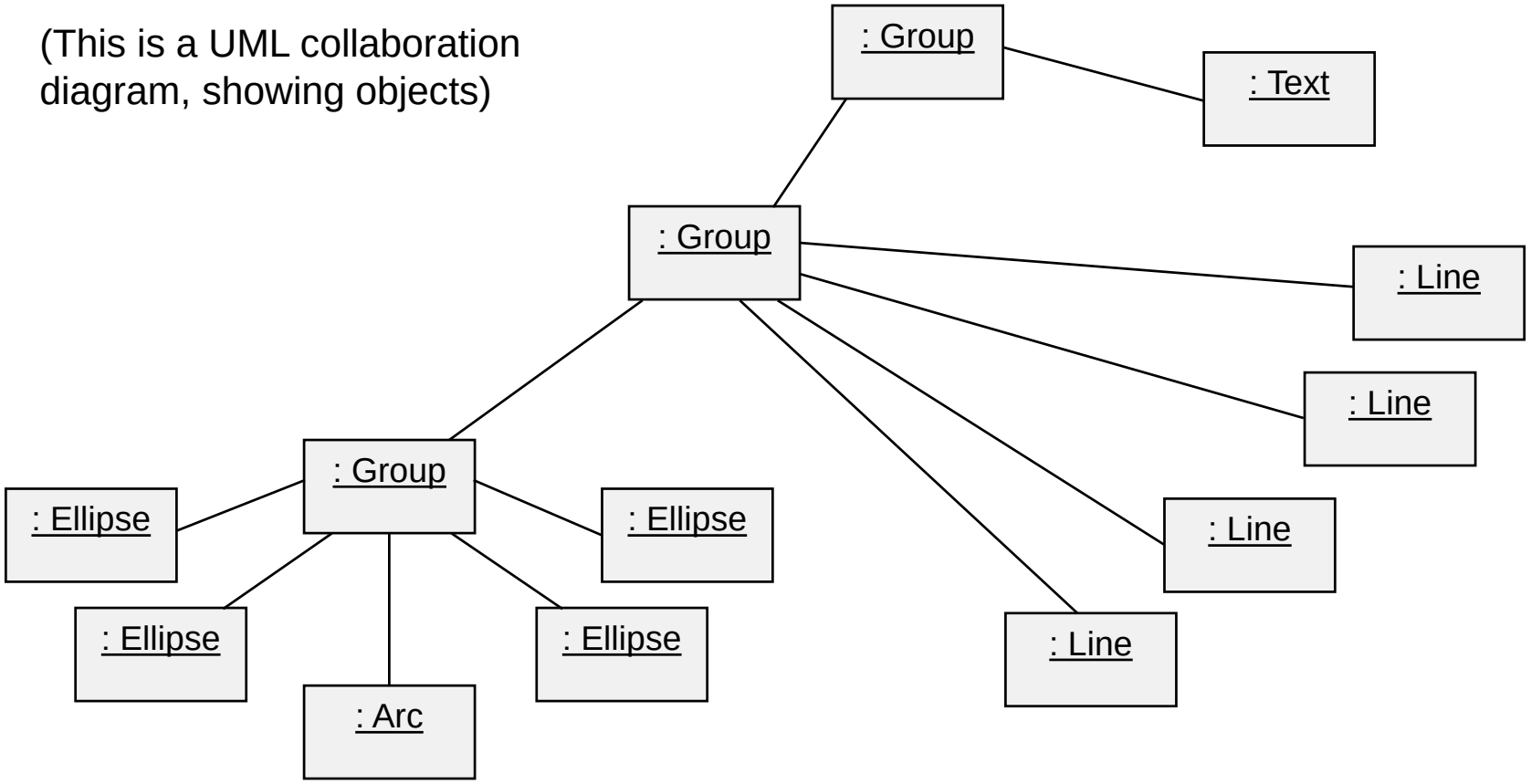


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Motivation

- **The software should mirror this structure**
  - Must allow the same operations on groups as on primitives

(This is a UML collaboration diagram, showing objects)



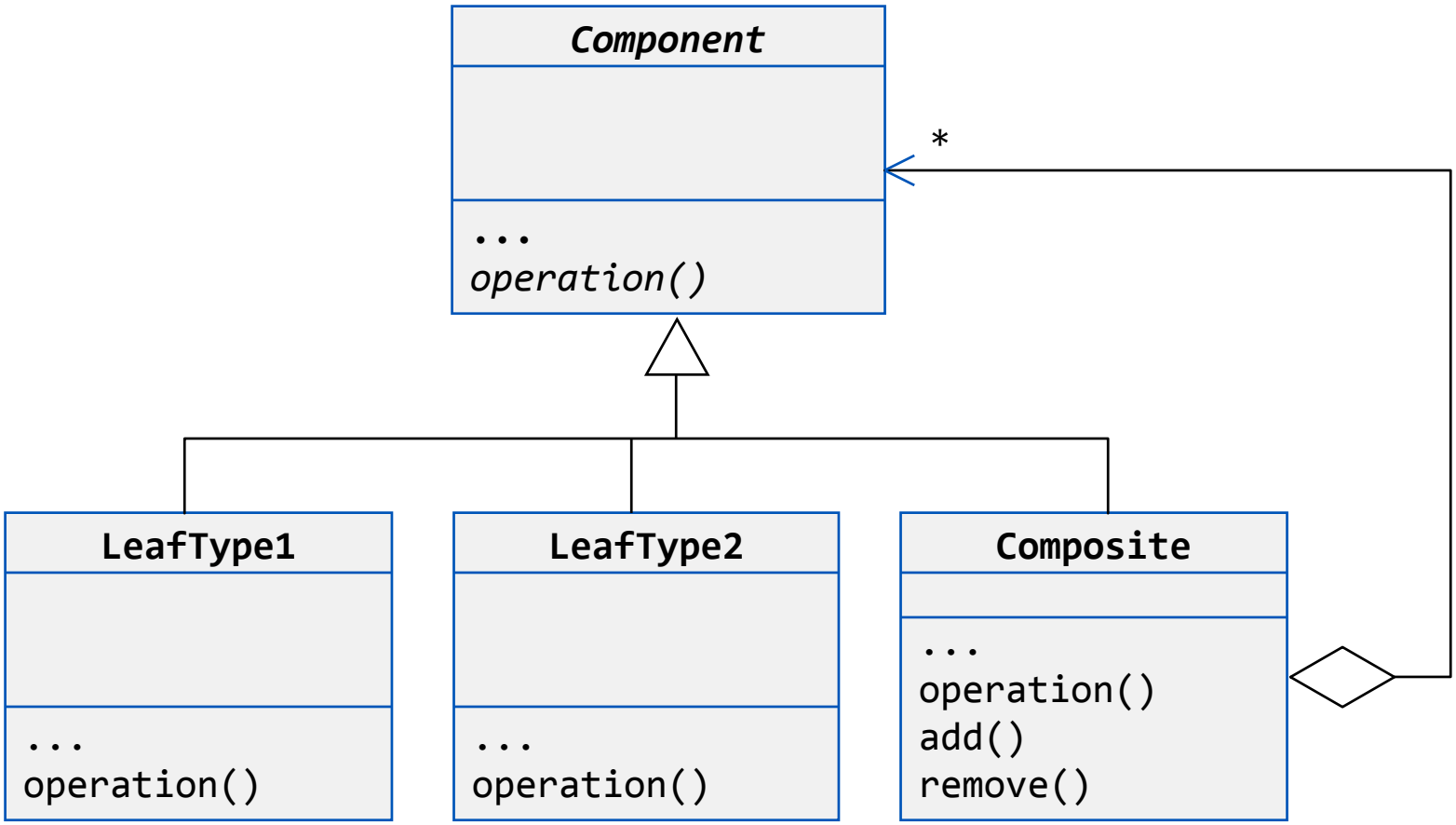
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Applicability

- **Use the *Composite pattern* when**
  - You want to represent a whole-part structure that may be arbitrarily deep
  - Clients should be able to ignore the difference between groups of objects and primitive objects

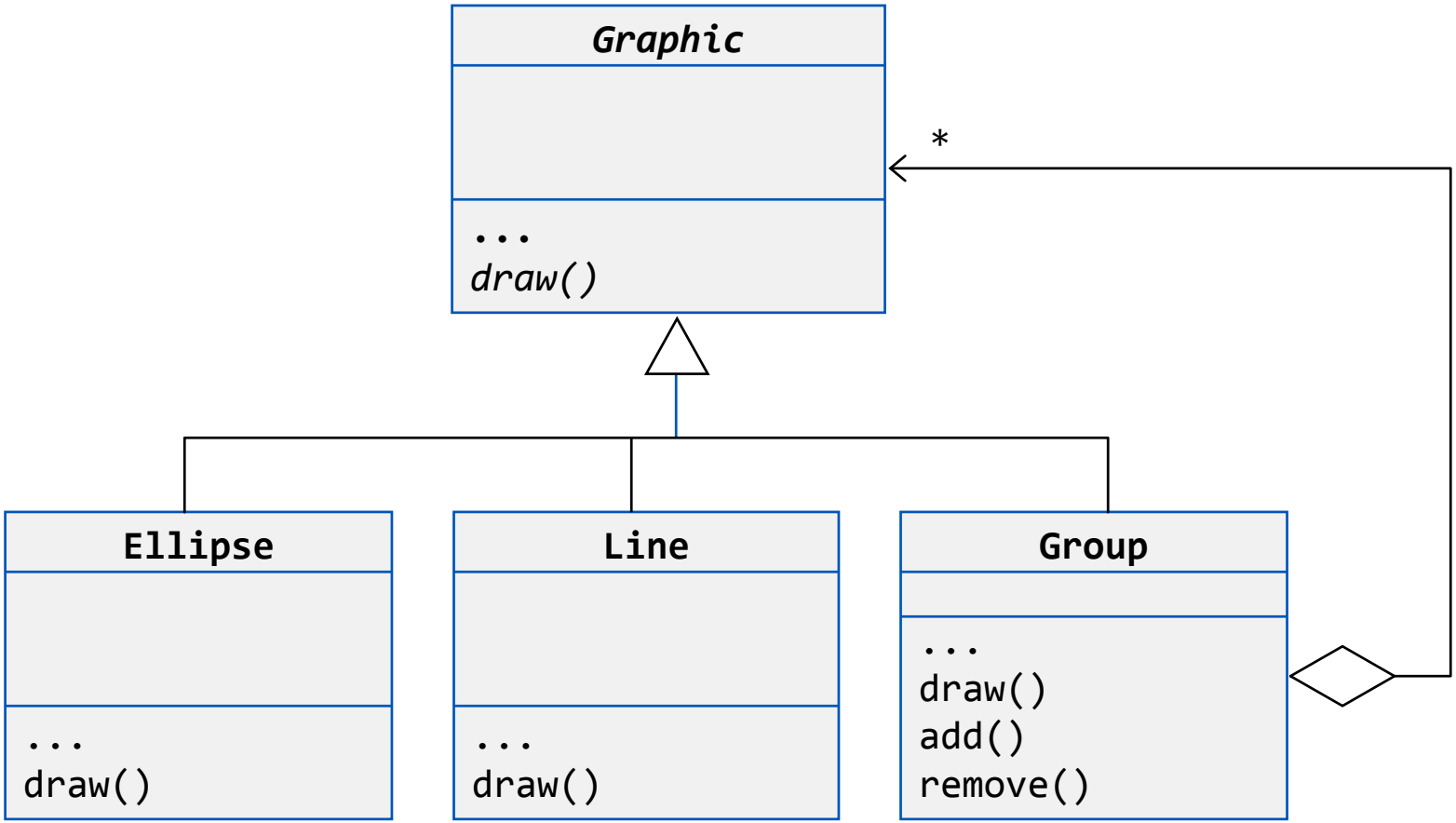
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Structure



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Structure



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Participants

- ***Component*** **(e.g., `Graphic`)**
  - Declares the interface for elements in the structure
- ***Leaf*** **(e.g., `Line`, `Arc`, `Ellipse`, `Text`)**
  - Implements behavior for a primitive object
  - A leaf has no children
- ***Composite*** **(e.g., `Group`)**
  - Maintains the collection of children
  - Implements the *child management* operations
  - Delegates operations to its children

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Collaborations

- **A client uses the component interface to manipulate an object in the structure**
- **A leaf object handles operations directly**
- **A composite passes requests on to each of its children**
  - Possibly performing additional behavior before and/or after delegating

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code

```
class Graphic{
public:
  virtual void draw() = 0;
  ...
private:
};
```

```
class Ellipse: public Graphic{
public:
  void draw();
  ...
};
```

```
class Group: public Graphic{
public:
  void add(Graphic*);
  void remove(Graphic*);
  void draw();
  ...
private:
  set<Graphic*> children;
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code

```
void Group::draw(){
        for (Graphic* g : children){
                g->draw();
        }
}

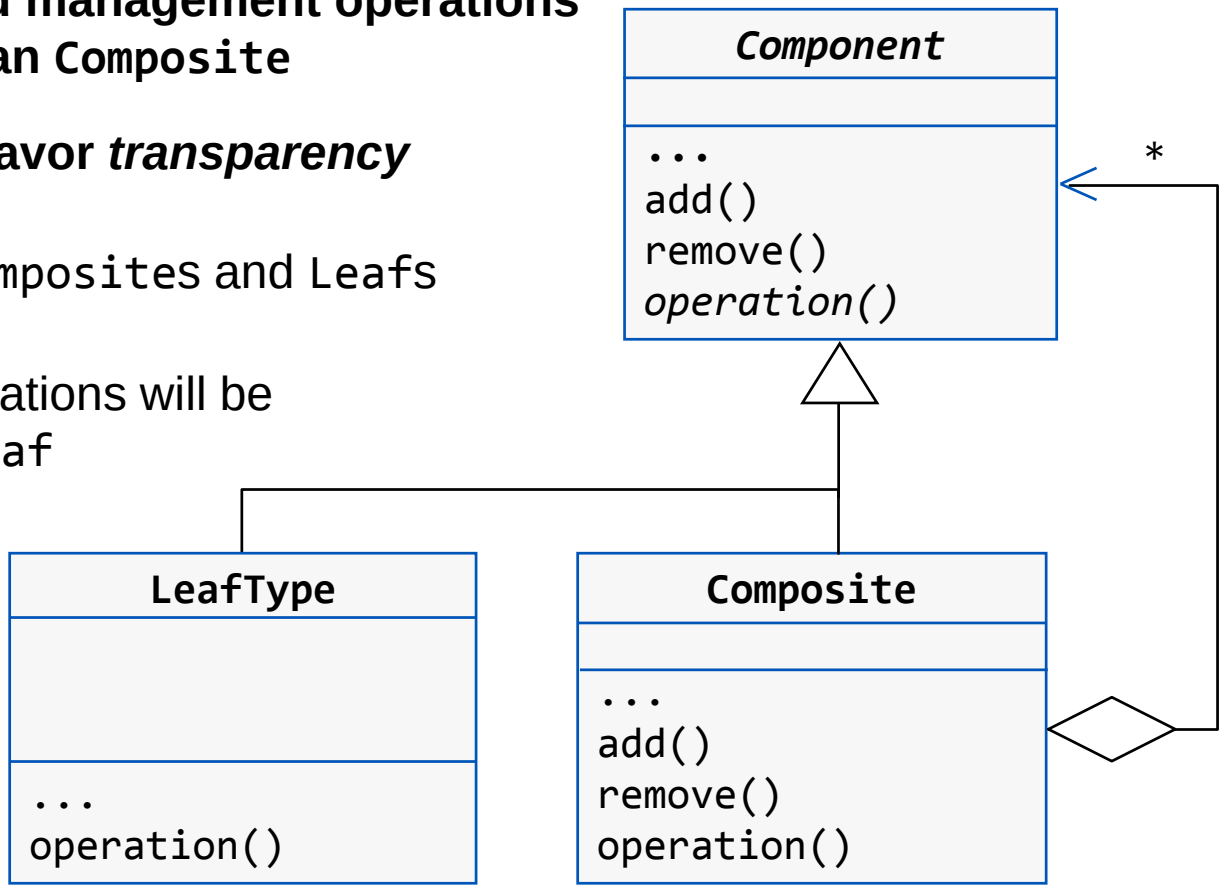
void Group::add(Graphic* g){
  children.insert(g);
}

void Group::remove(Graphic* g){
  children.erase(g);
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Consequences

- **Could declare the child management operations in `Component` rather than `Composite`**
- **This approach would favor *transparency* over *safety***
  - Client can handle `Composites` and `Leafs` uniformly
  - However, some operations will be meaningless for a `Leaf`
- **Often, a compromise is needed here**
  - A fat base class will have things derived classes don't want
  - Complex inheritance hierarchy will force special handling in code
  - Best tip: Favor fat abstract base classes



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Related Patterns: Flyweight

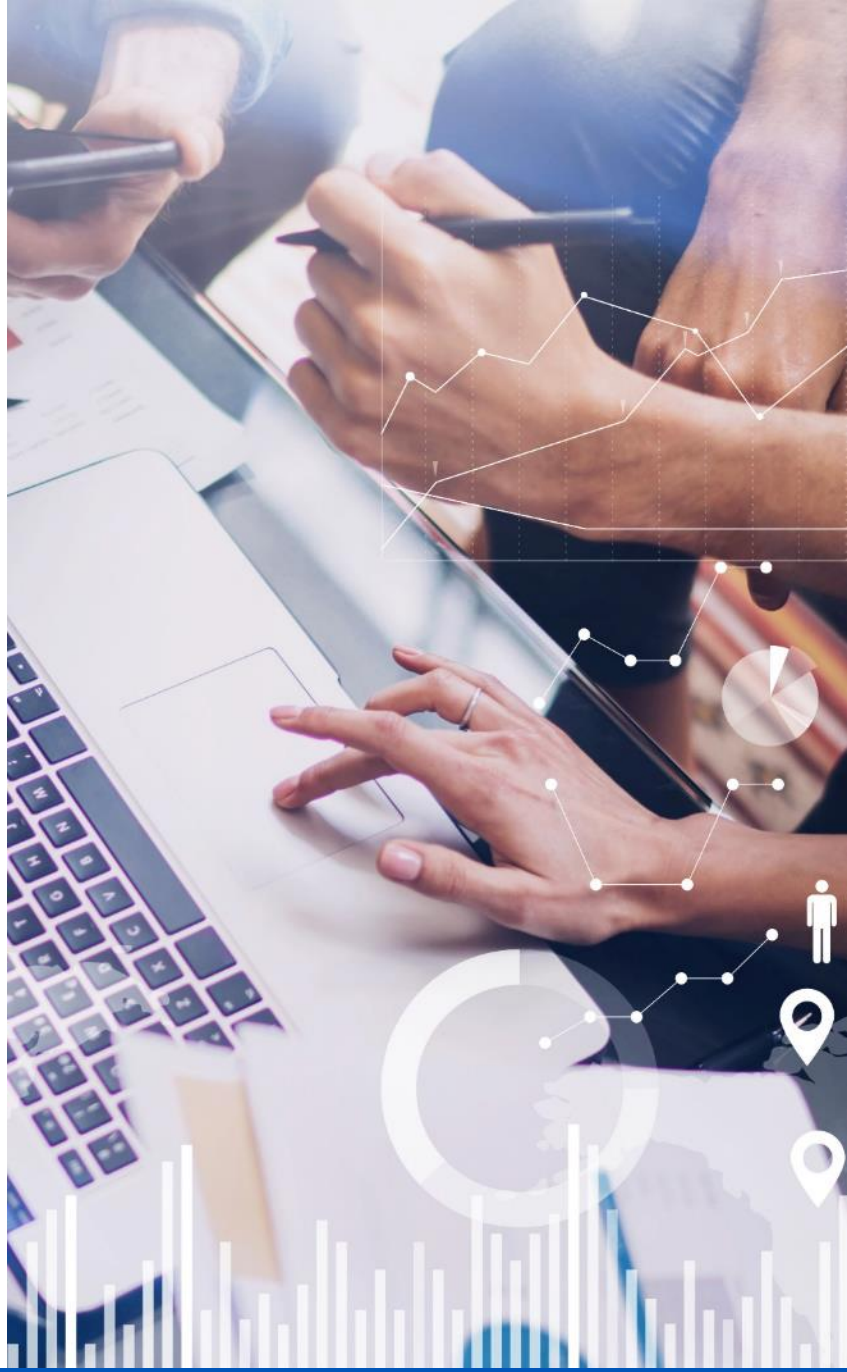
- **This pattern is used where the following is true**
  - An application uses a large number of objects that have much in common
  - This common state information can be made extrinsic
  - The *Flyweight* objects only contain what is special for them, and refer to the extrinsic common state
- **For example, a word processing document may contain many character objects, but far fewer distinct character/style combinations**
  - The Flyweight object only contains the characters
  - The extrinsic state objects are combinations of font, size, style, color, etc.
  - The extrinsic state objects can be organized in a hierarchical Composite structure with common state at the top and special versions as leafs

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

- Composite Pattern

Visitor Pattern



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

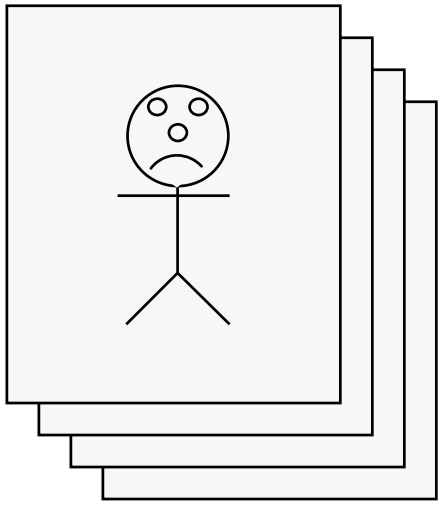
Identification

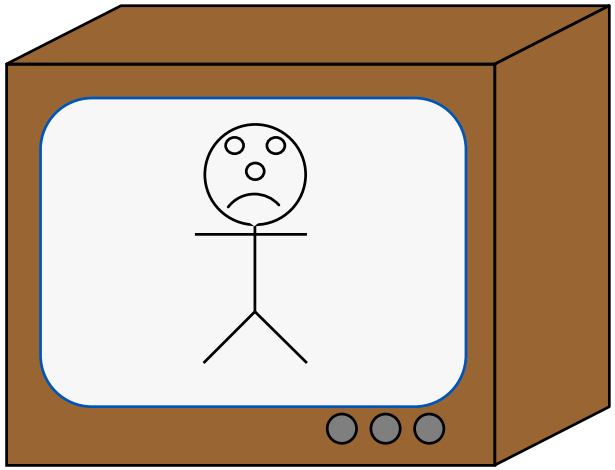
- **Name**
  - Visitor
- **Classification**
  - Behavioral
  - Object
- **Intent**
  - To represent an operation to be performed on the elements of an object structure
    - The structure is often a Composite
    - The data structure is stable and does not change often
  - The Visitor pattern lets you add new operations without changing the classes of the elements upon which they act
- **How many classes in the previous Composite example are affected when we add a new operation?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Motivation

- **In a graphics application, objects such as `Group`, `Line`, `Ellipse`, etc., hold the data that describes a picture**
  - Position, color, etc.
  - This data is independent of how the picture will be drawn
- **The behavior for drawing depends on the graphics tools used and the device on which the drawing is being done**





- **Where do we want to add the various operations for the different devices?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Motivation

- **We may also want to add operations (other than drawing)**
  - Still want to access all of the elements of the structure in the same order
  - For example, to ask the question, *"What is the total area of all filled shapes, so we can charge for the ink?"*
- **Putting the operations in the graphics classes would make it difficult to add new operations**
  - The operations will be spread out in many classes
- **Visitor pattern provides a well-defined way to separate algorithms from data**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Applicability

Use the Visitor pattern when:

- **An object structure contains many classes with differing properties**
  - For example, `Ellipse` might have a `getArea()` function, but `Line` would not
- **Many distinct operations must be performed on the objects in the structure**
- **The operations change more frequently than the classes that define the object structure**

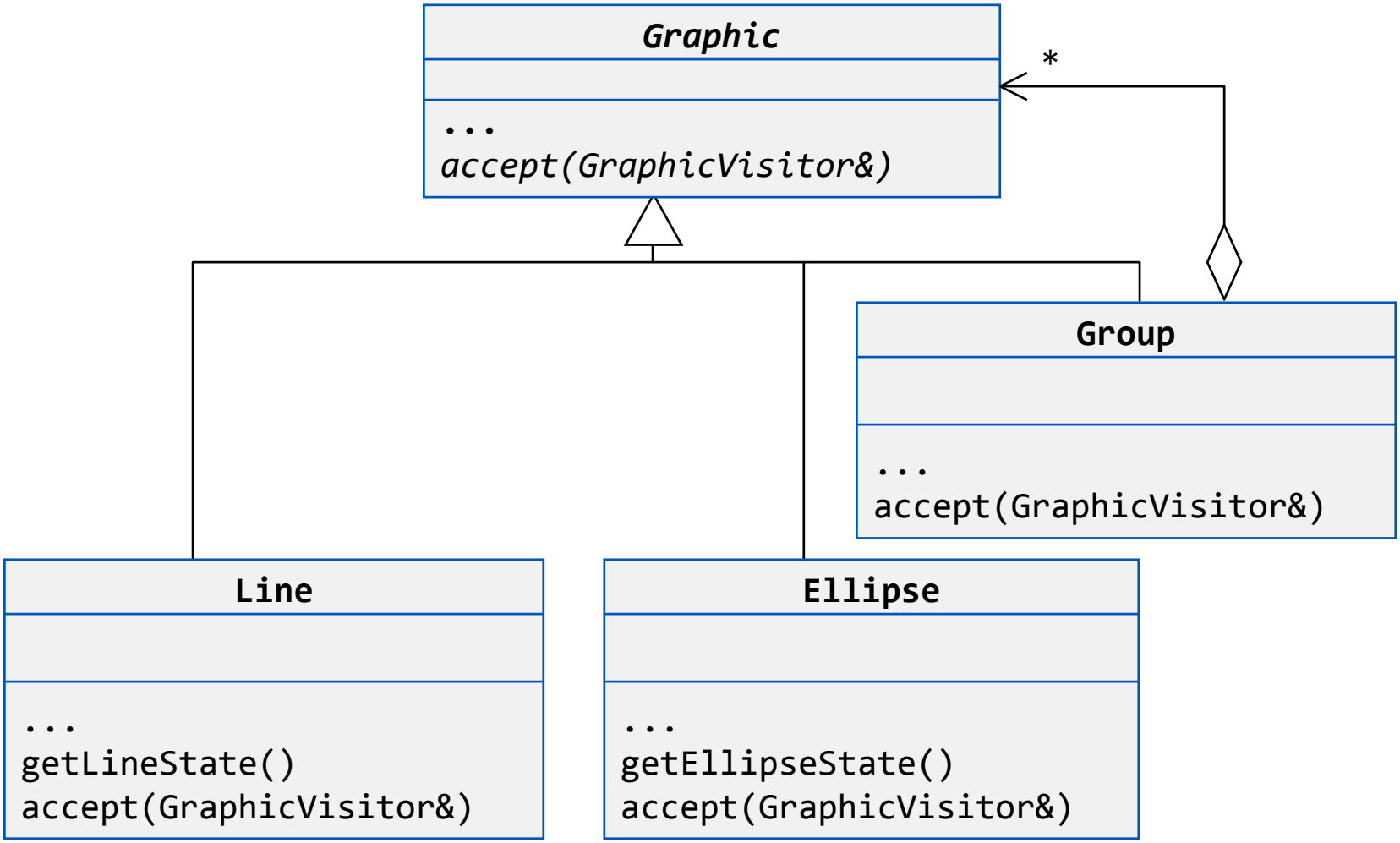
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Structure

- The *Visitor pattern* comprises two separate class hierarchies
- The *object structure hierarchy* comprises the classes of the objects that hold the data
- The *Visitor hierarchy* comprises the classes that define the operations to be performed on elements of the object structure

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Object Structure



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

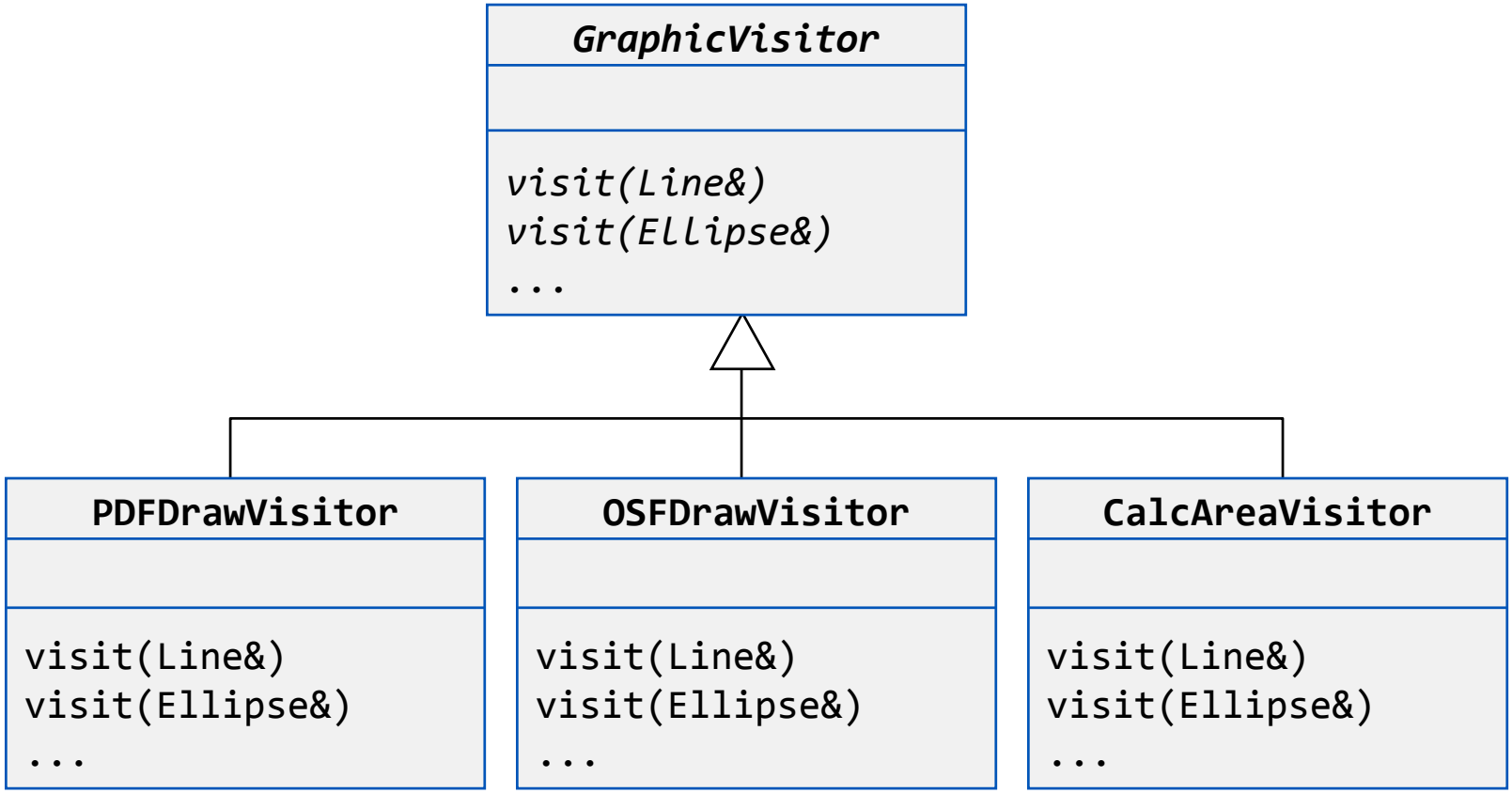
Participants: Object Structure

The object structure comprises the nodes that will be visited

- ***Component* node (e.g., `Graphic`)**
  - Declares `accept` taking a *Visitor* base class by reference
- ***Leaf* node (e.g., `Line`, `Ellipse`)**
  - Implements `accept`
  - Passes itself, by reference, to the Visitor's `visit` function
- ***Composite* node (e.g., `Group`)**
  - Delegates `accept` to each of its children

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Visitor Hierarchy



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Participants: Visitor Hierarchy

- ***Abstract Visitor*** **(e.g., `GraphicVisitor`)**
  - Declares a set of pure virtual functions—one for each type of leaf node to be visited
    - Each function accepts a particular leaf node type, by reference, as its parameter
    - Can use overloading, with all functions called `visit`, or can be named individually; e.g., `visitLine`, `visitEllipse`
- ***Concrete Visitor*** **(e.g., `PDFDrawVisitor`, etc.)**
  - Defines each of the `visit` functions appropriately
  - A `visit` function will typically request the state of its parameter node and act on it accordingly

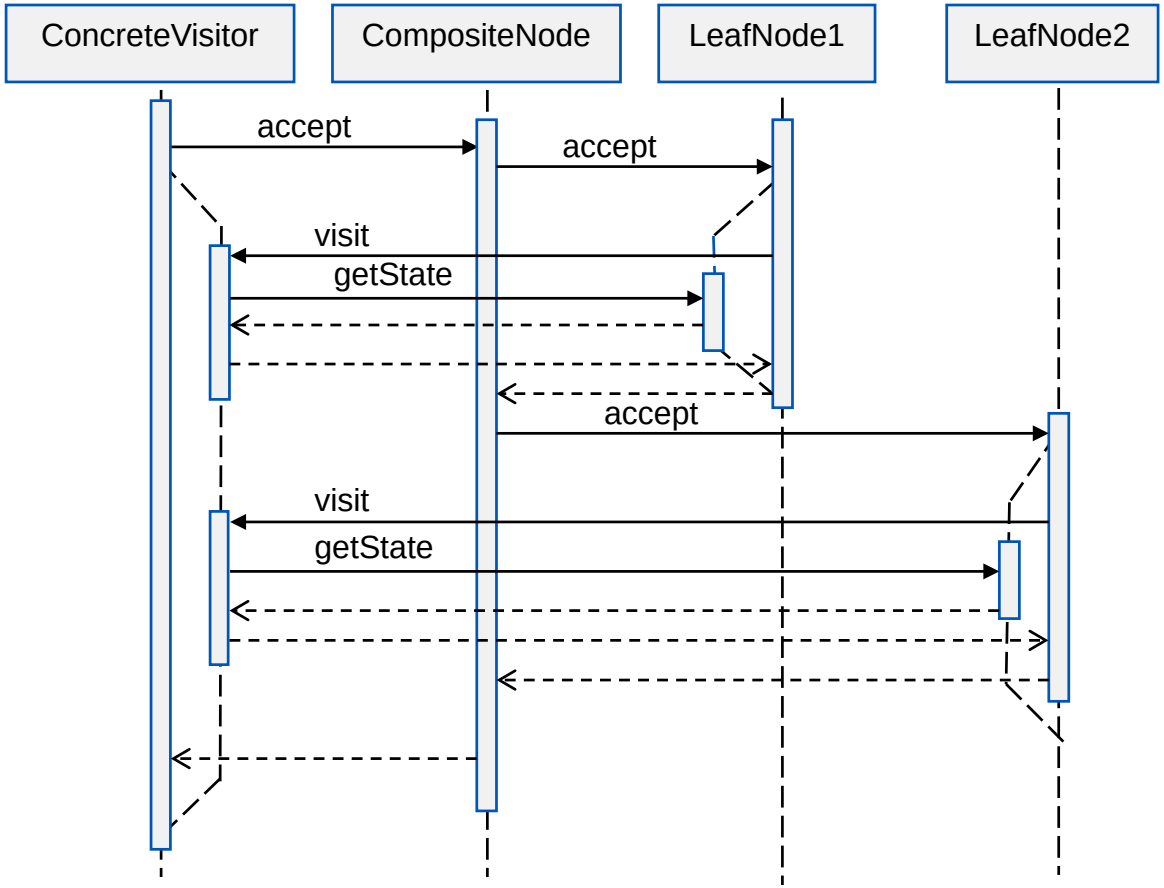
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Collaborations

- **To use the Visitor pattern, a client creates a concrete Visitor**
- **The concrete Visitor is passed as the parameter to the object structure's `accept` function**
  - A Composite's `accept` function calls `accept` for each of its children, passing on the Visitor as the parameter
  - A leaf node's `accept` function passes itself as parameter to the Visitor's appropriate `visit` function
- **This is an example of *double dispatch***
  - The call must be polymorphic, based on two quantities
  - Which `visit` function is called depends on the class of both the Visitor and the node being visited
  - C++ polymorphism inherently supports only *single dispatch*
    - The function called depends only on the class of the receiving object

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Visitor Pattern: Sequence Diagram



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code

```
class Graphic {
public:
...
  virtual void accept(GraphicVisitor& v) = 0;
};
```

```
class Ellipse: public Graphic {
public:
  void accept(GraphicVisitor& v){
    v.visit(*this);
  }
  int getMajorAxis();
...
};
```

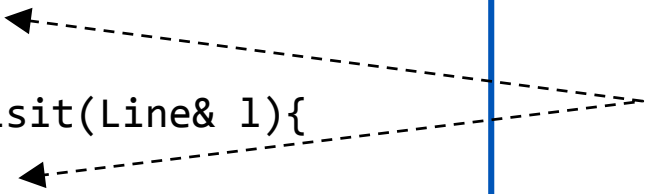
```
class Group: public Graphic {
public:
  void accept(GraphicVisitor& v);
  ...
private:
  set<Graphic*> children;
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sample Code

```
void Group::accept(GraphicsVisitor& v){
        for (Graphic* g : children){
                g->accept(v);
        }
}
```

```
void PDFDrawVisitor::visit(Ellipse& e){
  int major = e.getMajorAxis();
  ...
}
void PDFDrawVisitor::visit(Line& l){
  ...
}
```



Gather state, then draw using PDF facilities

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Consequences

- **Adding new operations is easy**
  - Define a new concrete Visitor class
- **Adding new node types to the object structure is difficult**
  - Every class in the Visitor hierarchy must be modified to visit the new node type
- **Does the Visitor pattern break encapsulation?**
  - A node type must provide a Visitor with access to its state, so encapsulation is weakened
  - However, the increased flexibility of the Visitor pattern often makes this an acceptable compromise
  - Consider the original alternative: adding code for each operation in every data structure class

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Hands-On Exercise 7.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 7.1: Composite and Visitor Patterns or Hands-On Exercise 7.1A

In this exercise, you will be working on a simple Address Book application (a "Hello, World"-type Visitor exercise)

- You will explore a typical Composite data structure
  - Address Book entries and address groups, including groups of groups
- You will complete the implementation of a Composite operation
- You will add new operations to the Address Book with the Visitor pattern
- Exercise 7.1

There is an alternative exercise that uses Visitor for the `Pole` class of the pelican crossing project (a more realistic Visitor exercise)

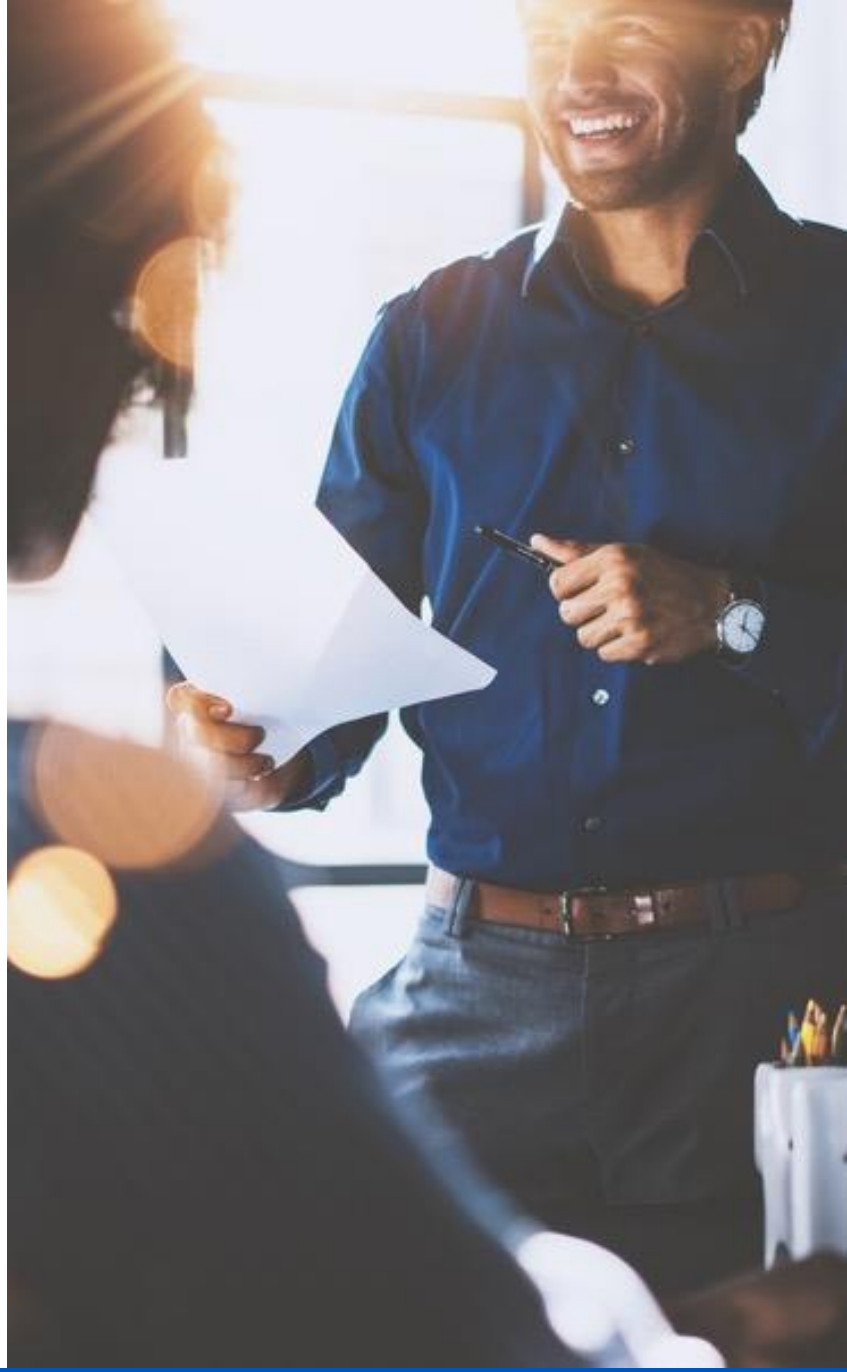
- It will separate the Pole from possibly different ways to display it using the Visitor pattern
- Exercise 7.1A

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

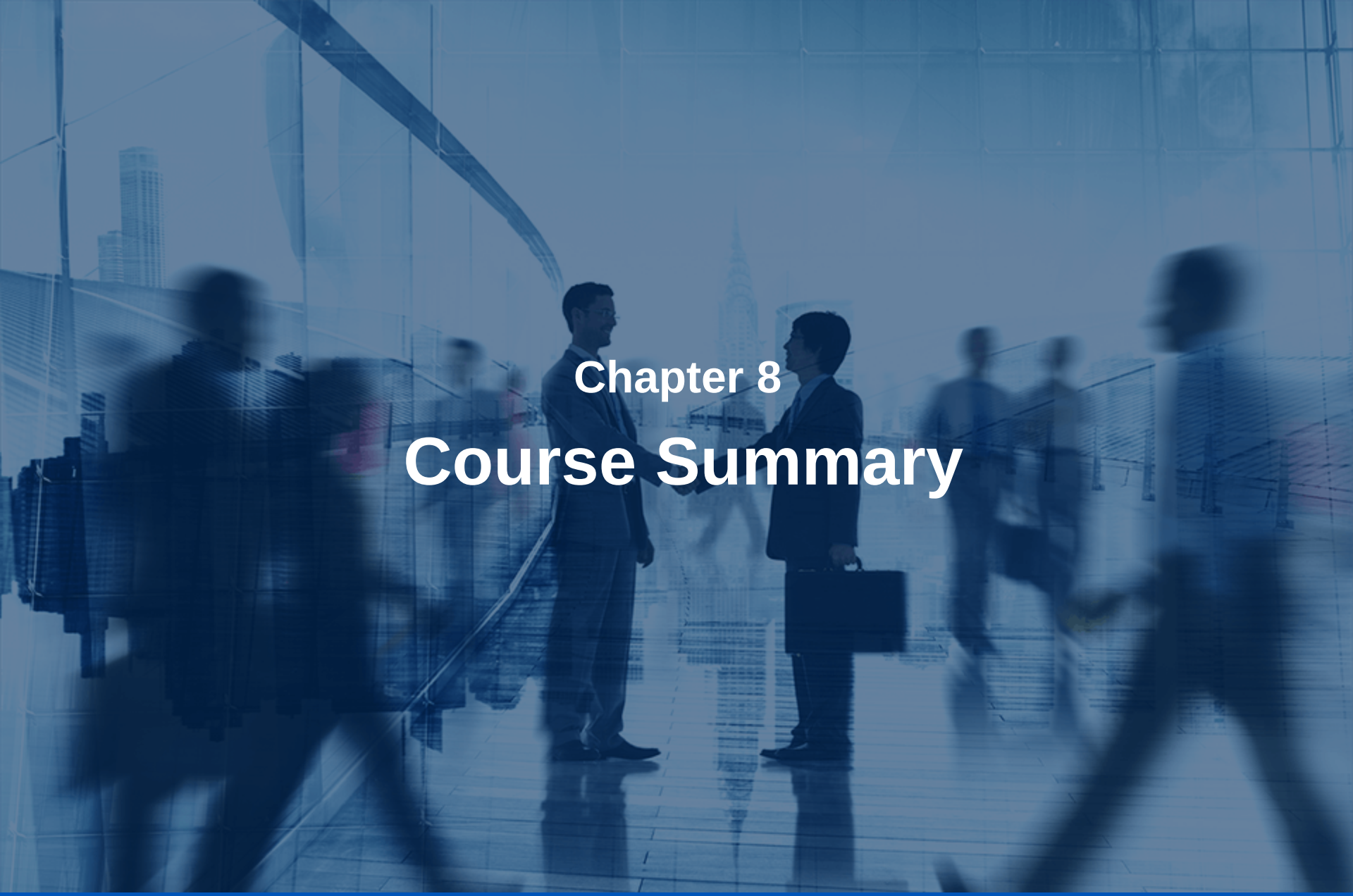
Objectives

Learn about patterns that are often used together

- **The Composite pattern**
  - A pattern for simplifying the handling of recursive, whole-part structures
- **The Flyweight pattern**
  - A pattern that permits efficient handling of structures with a large number of objects with common state
- **The Visitor pattern**
  - How to separate operations from state in situations where they may need to change independently



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



Chapter 8

Course Summary

LEARNING TREE™ INTERNATIONAL

Course Objectives

- **Use all significant features of the C++ language, including the latest C++ standard**
- **Apply design patterns to produce high-quality object-oriented C++ code**
- **Make effective use of libraries**
  - The Standard Library
  - Boost open-source libraries



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Advanced C++ Features

- **Exceptions**
  - `try`, `throw`, and `catch`
  - Tips for effective exception handling
- **RTTI and casting**
  - Determining runtime type
  - Type-safe casting
- **Other features covered as “asides”**
  - Overloading `new` and `delete`
  - `explicit`, `mutable`, and `const`
  - `namespaces`
  - Templates
  - Memory layout
  - `virtual` destructors



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

C++11/14/17/20 and Design Patterns

- **New features in the C++ standards**
  - Lambdas, auto declarations, new initialization syntax (including lists), r-value references, and move semantics
  - Controlling default constructors and assignment with `=default`/`=delete`
  - New libraries: threads, timing, extensions to the containers
- **Design patterns**
  - Singleton
  - Adapter
  - Template Method
  - Observer
  - Composite
  - Flyweight
  - Visitor
  - Factory

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Effective Use of Libraries

- **The Standard Library**
  - Collections
  - Strings
  - Streams
  - Threads
- **Storage libraries**
  - Boost serialization facilities



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Future



- This course has given you an insight into the best practices of C++ programming
- You have learned a new way of thinking about programming, as well as many new techniques

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A

Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Description of Pelican Crossing



The name pelican is used to describe a kind of pedestrian crossing found in the United Kingdom. The name is derived from a contraction of the term PEdestrian LIght-CONtrolled. As there is no such thing as a pelicon, it has come to be called a pelican.

A pelican crossing is neither used by, nor does it look like, a pelican. Contrast this with a zebra crossing, which, while also not being used by its eponymous animal, does look like one—at least in that it has black and white stripes.

A pelican crossing presents red, yellow (or amber), and green lights to oncoming traffic. It presents a red, standing-person-shaped light and a green, walking-person-shaped light to pedestrians. There is also a button for pedestrians to push, which will cause a light marked “WAIT” to illuminate if the pedestrian cannot cross immediately. There is normally a pole equipped like this on either side of the road.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

In its simplest form—the form we will be simulating—the crossing has no form of traffic sensor. Many real pelican crossings do have traffic sensors.

The light sequence is as follows:

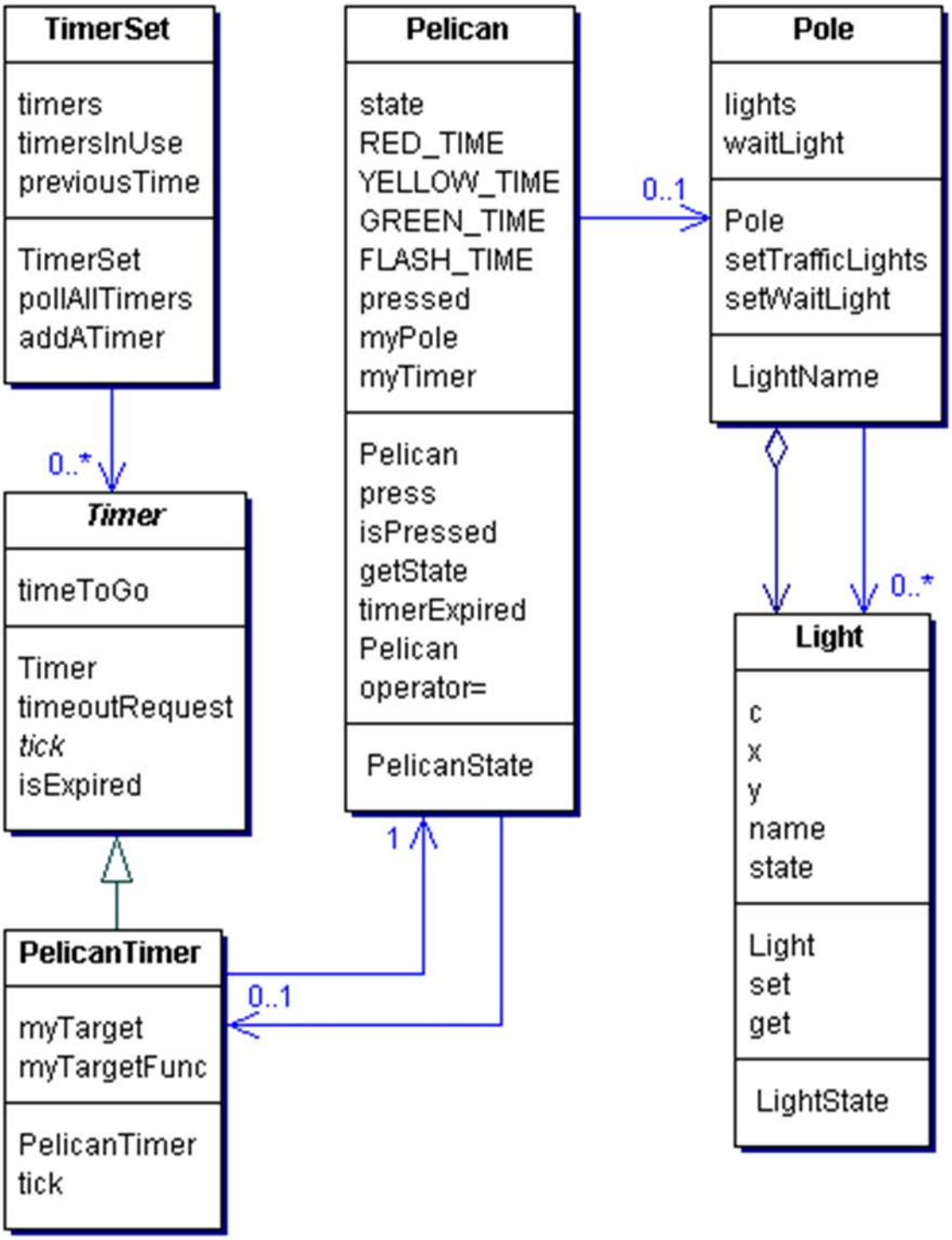
Light seen by pedestrians	Light seen by traffic	Meaning for pedestrians	Meaning for traffic
Red, standing	Green	Don't walk	Go
Red, standing	Yellow	Don't walk	Stop
Green, walking	Red	Walk	Stop
Green, flashing	Yellow, flashing	Don't start to walk but continue if already crossing	Stop if pedestrians are crossing; otherwise go

When a pedestrian pushes the button, the WAIT light illuminates and the lights start their sequence. When the traffic lights show red, the WAIT light goes out. When the lights go back to green, traffic is allowed to flow for a period. If a pedestrian pushes the button during this time, the WAIT light will illuminate right away. However, the lights will not begin to sequence immediately.

There is usually an audible signal that coincides with the pedestrian Walk state. It is an intermittent beep, intended to help the visually impaired to use the crossing.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Pelican Crossing Program: Class Diagram



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Pelican Crossing Program: Class Descriptions

`TimerSet`
This class holds a collection of `Timer` objects. The program initially has only one `Timer` in this collection, but you may add more as you develop it. The function `pollAllTimers()` is called frequently by `main()`. The function calls `tick()` for each `Timer` instance each time a second has elapsed. In Exercise 2.1, you will convert this class into a *singleton* to ensure that the program contains a single collection of timers.

`Timer`
The `Timer` class provided a general-purpose countdown-timer facility. To use this facility, an object that wants to receive timeout notification derives a class from `Timer`, in which the member function `tick()` is overridden. This function should call the member function `isExpired()`, which returns `true` when the required period has elapsed.

`PelicanTimer`
This class is the derivation of `Timer` specifically for use by class `Pelican`.

`Pelican`
Class `Pelican` represents the crossing as a whole. It has a member function `press()` that is called by `main()` when the space bar is pressed. A `Pelican` cycles through its sequence upon timer expiries. A `Pelican` has an object of class `Pole` that displays the lights to the user.

`Pole`
A `Pole` has an array of `Light` objects representing red, yellow, green, red-person, and green-person. It also has a light (not part of the array) representing the WAIT light. Class `Pole` has member functions `setTrafficLights()`, and `setWaitLight()` that are called by the `Pelican` when it changes state.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Pelican Crossing Program: Class Descriptions

`Light`

A `Light` has data members defining a state (on, off, or flash), a color, a screen position, and a name (e.g., WAIT or *|-<). A `Light` displays itself in a console window using colored characters.

`main()`

A single object of class `Pelican` is defined within `main()`. `main()` performs a busy loop, using `Keyboard::isCharAvail()` to determine if a key has been pressed. If a key has been pressed, it checks whether it is the space bar. If it is the space bar, `Pelican::press()` is called. If no key is pressed, `TimerSet::pollAllTimers()` is called; this is the function that the timer mechanism uses to determine when a second has elapsed.

Using the program

Press the space bar to change the lights and press `<Q>` to quit.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Running the Pelican Under Unix/Linux

The steps here were developed and tested under Ubuntu 64-bit version 14.0. The code has also been successfully tested on MacOS X, and should work on any Unix/Linux version.

This process below is based on a clean installation and is using Eclipse as the build tool. Some steps may be skipped if, for example, Eclipse has already been installed. If you are familiar with other C++ development environments, like Apple's XCode, or just want to build directly from the command line, that will also work.

If Eclipse and GCC are already running successfully, but the Pelican will not build, then it is likely that the Curses library has not been installed.

Installing JVM

Eclipse uses Java and requires a Java Virtual Machine to run.

- Open the terminal window:
  <ctrl><alt><t>
- Update the Advanced Packaging Tool package index:
  `sudo apt-get update`
- Install Java:
  `sudo apt-get install default-jre`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Installing GCC

The GCC C++ compiler and related tools used by Eclipse require separate installation.

- Still at the terminal window, type:
  `sudo apt-get install g++`
- Now check the compiler version
  `g++ --version`
- The software requires version 5 or later for the full C++14 features
- If the version is earlier than 5, type the following
  `sudo add-apt-repository ppa:ubuntu-toolchain-r/test`
- And again call
  `sudo apt-get update`
- Then install the specific version
  `sudo apt-get install g++-5`
- After the install, the command g++ will still point to the older version. Update it with
  `sudo update-alternatives --install`
  `/usr/bin/g++ g++ /usr/bin/g++-5 20`

Installing Curses

The Pelican program uses the Curses library to manage terminal output.

- Still at the terminal window, type:
  `sudo apt-get install libncurses5-dev`

Installing Eclipse

While C++ programs may be built directly from the terminal window, it is usually easier to use Eclipse CDT (C/C++ Development Tooling) as an IDE

- Download the appropriate Eclipse version from:
  http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/mars1
- When asked, open with Archive Manager
- Right-click on the downloaded file and extract to a folder of your choice

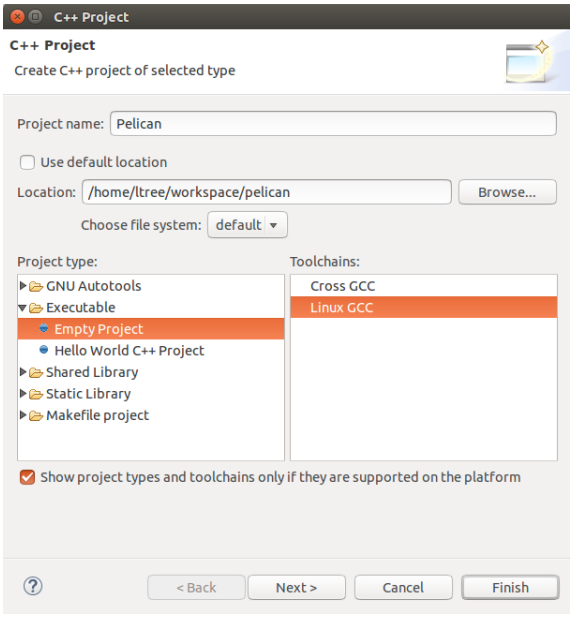
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Creating the Project

You are now ready to create the Pelican project.

- Double-click the Eclipse icon to run the IDE
- Create an empty project with
  File > New > C++ Project
- Fill in the dialog as below, choosing your own location path
  Make sure you select Linux GCC for the toolchain



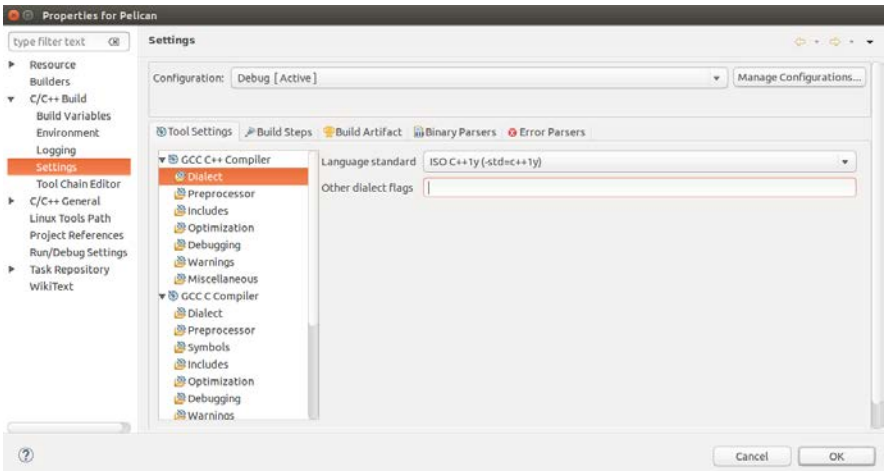
- Click Finish (other options may remain as default)
- Copy all the source files (.h and .cpp) from the Pelican exercise folder into the folder created
- Copy all the files from the DOS_Unix folder *except* Keyboard_DOS.cpp and Screen_DOS.cpp into the newly created project folder on your Linux system
- In Eclipse, right-click on the Pelican project and click Refresh to see the files displayed

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

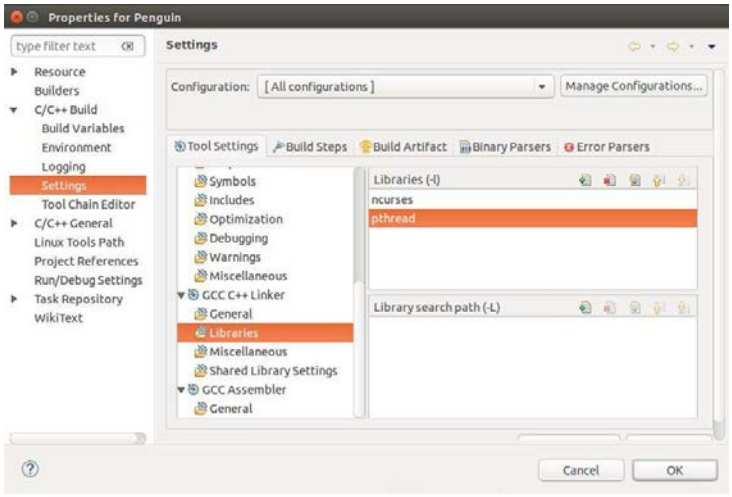
Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Configuring the Build

- Right-click on the Pelican project and select Properties
- Navigate to C/C++ Build > Settings > GCC C++ Compiler > Dialect
- Select the latest version of the compiler



- In the same dialog navigate to GCC C++ Linker > Libraries
- In the Libraries panel click the '+' button and add 'ncurses' to the list of libraries
- Also add 'pthread' to the libraries



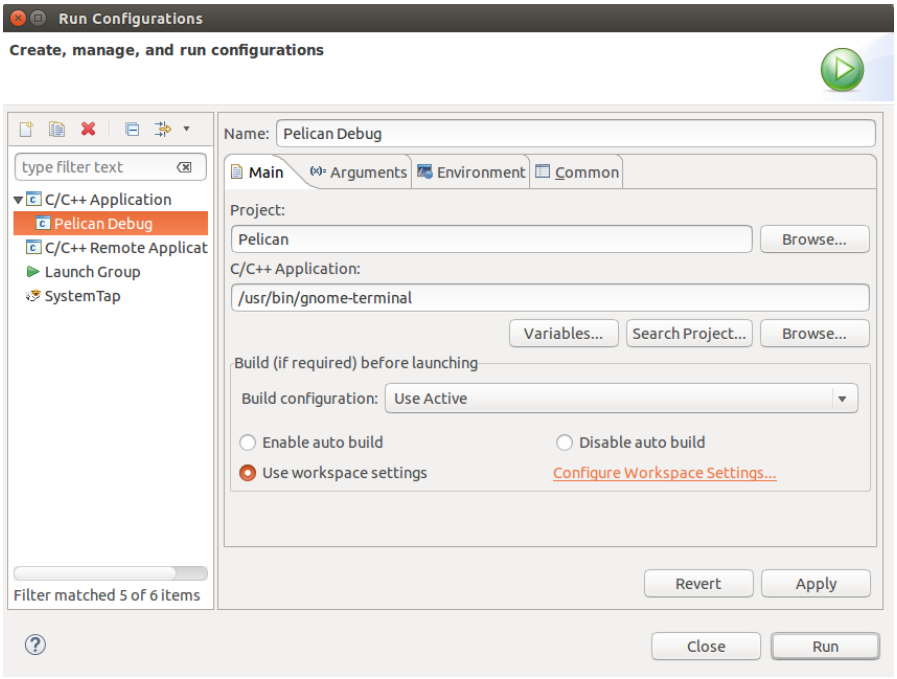
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

- The project is now ready to build. Right-click on the Pelican project and click Build. You may get warnings, but there should be no errors. Do NOT attempt to run the project yet.

Configuring output

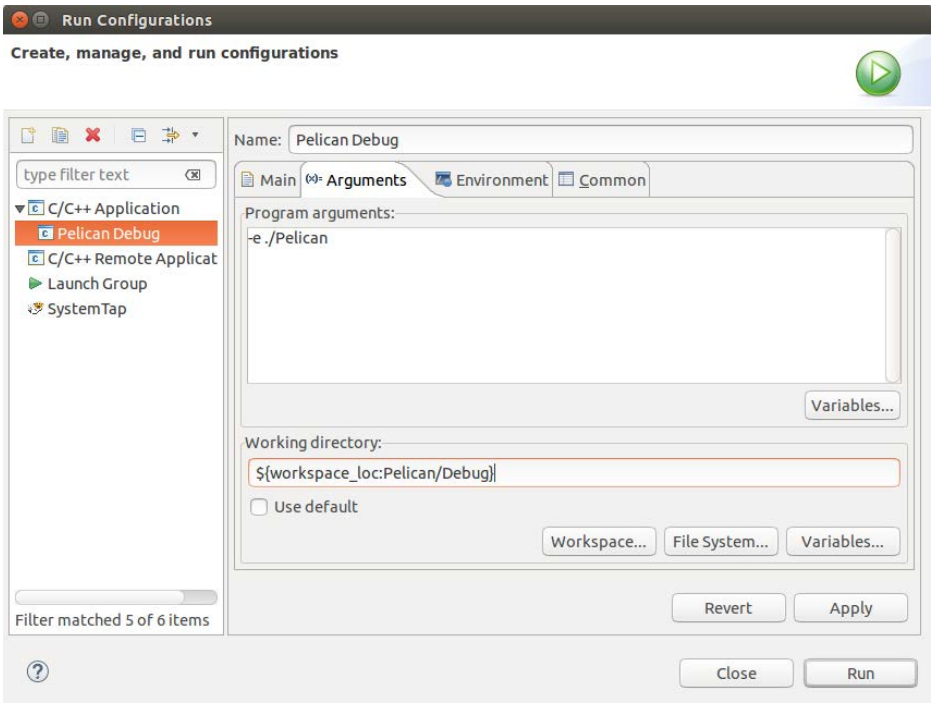
By default, Eclipse uses a built-in console that is not compatible with the Curses library. We need to run the Pelican in the terminal window. We can do this simply by launching a terminal window (<ctrl><alt><t>) and running the program. However, it is more convenient to configure Eclipse to launch the window.

- Right-click on the Pelican Project and select Run As > Run Configurations…
- Add a new launch configuration with the '+' just above the filter box.
- In the C/C++ Application text box type the following to launch the terminal:
  `/usr/bin/gnome-terminal`



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

- Navigate to Arguments tab of the same dialog
- In the Program arguments text box type the following to run the Pelican within the terminal:
  `-e ./Pelican`
- Further down, uncheck ‘Use Default’ and change the Working Directory to:
  `${workspace_loc:Pelican/Debug}`



- Click ‘Apply’
- Click ‘Run’ to launch the Pelican

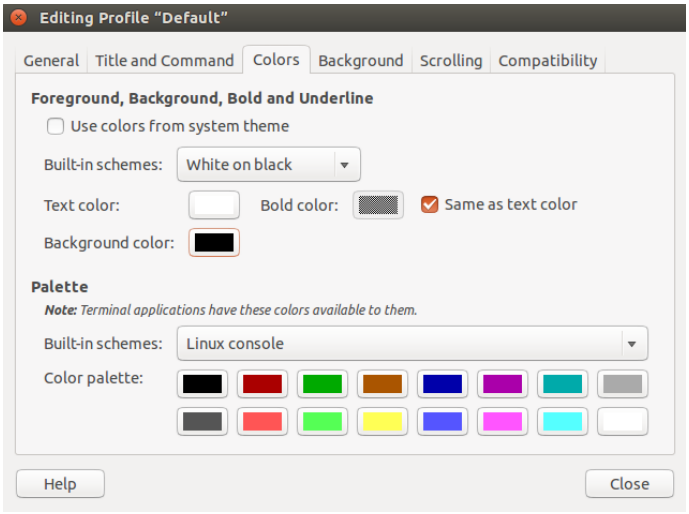
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix A: Description of Pelican Crossing: Running the Pelican Program on Unix/Linux

Configuring the Terminal

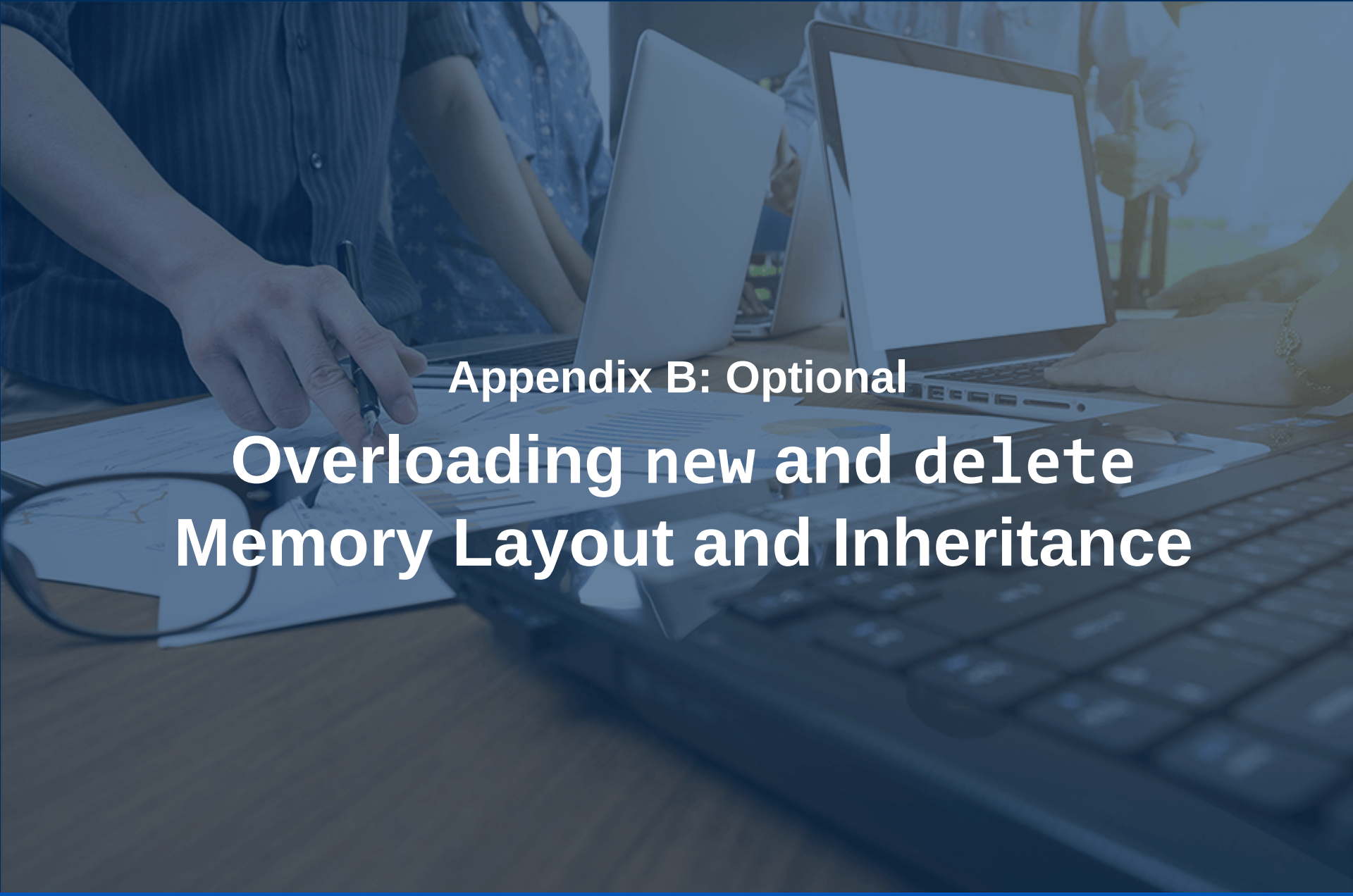
The default terminal background color may not be the black used by the Pelican application. This can be fixed as follows.

- Right-click anywhere on the terminal and select Profiles > Profile Preferences
- Uncheck ‘Use colors from system theme’
- Under ‘Built-in schemes’ select ‘White on black’
- In the Palette section, under ‘Built-in schemes’ select ‘Linux Console’



Congratulations! You can now run the Pelican program under Linux.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.



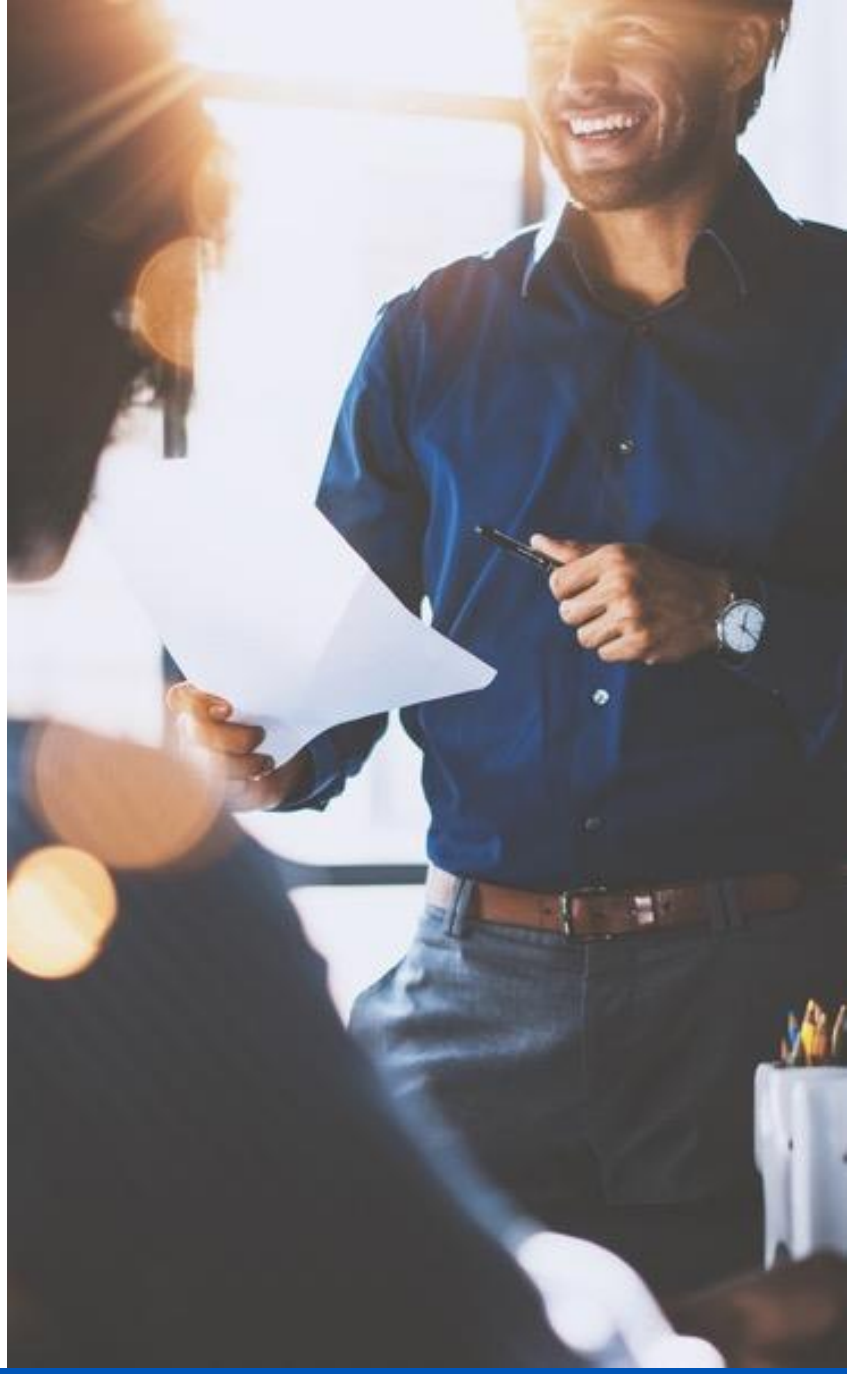
Appendix B: Optional

Overloading `new` and `delete` Memory Layout and Inheritance

LEARNING TREE™ INTERNATIONAL

Objectives

- **This appendix contains optional material that may be covered depending on interest and timing**
  - There will be no exam questions on this section
- **It expands two topics**
  - Overloading `new` and `delete` for detailed memory allocation handling
  - Memory layout issues and details of inheritance, virtual tables, multiple and virtual inheritance



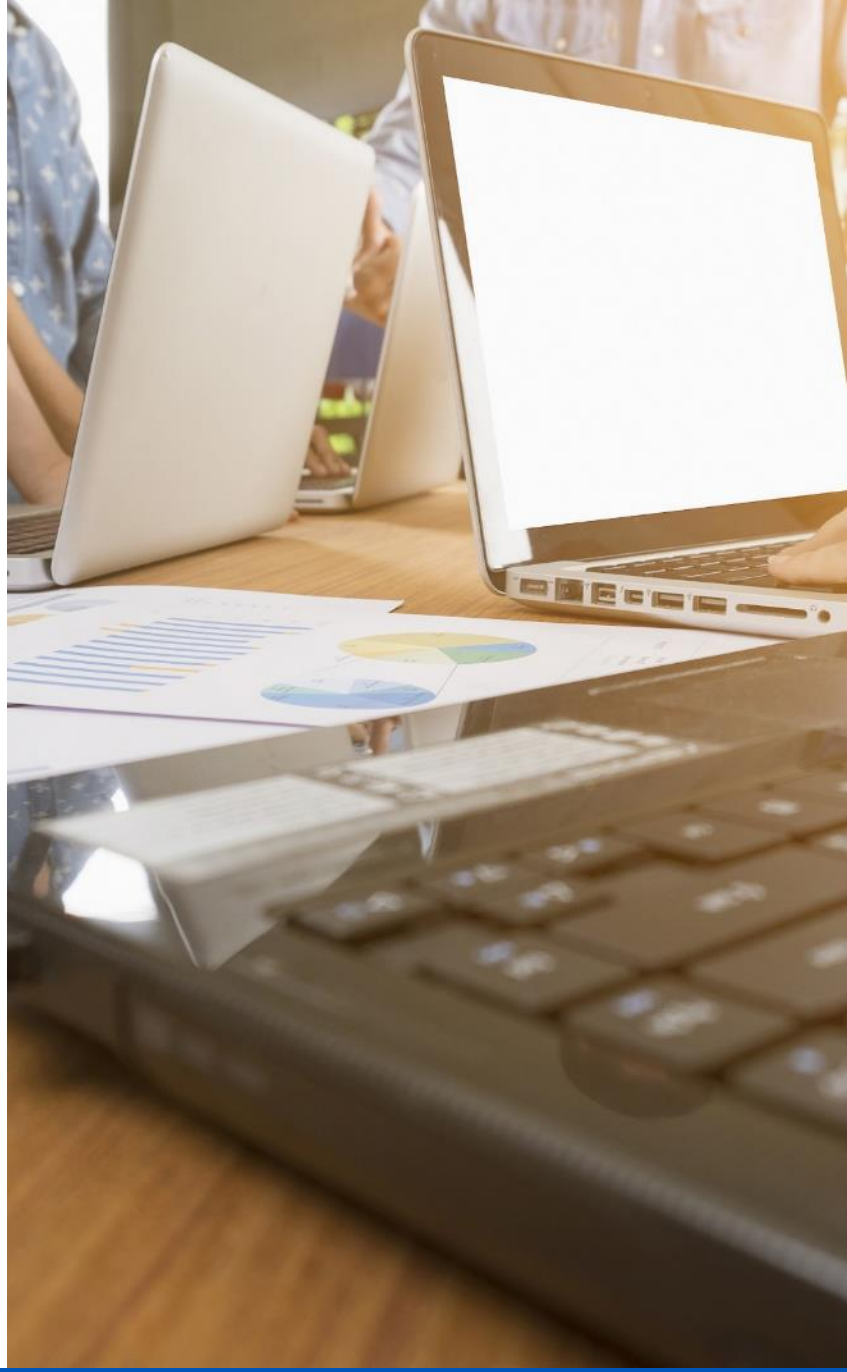
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Memory Layout and Derivation

Introduction

- Single Derivation
- Multiple Derivation
- Virtual Derivation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

new

- **Objects are created dynamically using a *new expression***
  - `MyClass* myPtr = new MyClass;`
- **The `new` expression does the following**
  - Invokes `operator new`, which returns the address at which the object will be built
  - Invokes a constructor to initialize the object
- **The default `operator new` returns an address from heap storage**
  - The programmer can overload this operator to take control of dynamic object creation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Overriding new

- **The default `operator new` has the following signature**
  - `void* operator new(size_t n);`
    - Where `n` is the number of bytes to be allocated for the object
    - The `new` expression supplies this value
- **This operator can be overridden for a specific class**
  - For example, you may want objects of a specific class to be allocated from fast or shared memory

```
class MyClass{

  ...
  void* operator new(size_t n){
    void* address;
    address = ... // an appropriate address
    return address;
  }
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Overloading new

- **Other information can be passed to your `operator new`**
  - For example, you might want to pass as an argument the address at which the object is to be allocated
  - This is done by adding an extra parameter to `operator new`

```
class MyClass{
  ...
  void* operator new(size_t n, void* address){
    ... // may not be necessary to do anything else
    return address;
  }
};
```

- **This overloading is invoked by a *placement* `new` *expression***
  - What is happening here?

```
MyClass * p = new(0xFFFF) MyClass;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Allocating Arrays

- **`operator new` allocates a single object**
- **To allocate an array of objects, the separate `operator new[]` is used**
  - `operator new[]` must be overloaded or overridden explicitly
  - Which operators get called in the following code fragment?

```
class MyClass{

  ...
  void* operator new(size_t n);
  void* operator new[](size_t n, int x);
};
```

```
int main()
{
  MyClass *ptr1 = new MyClass;
  MyClass *ptr2 = new MyClass[4];
  MyClass *ptr3 = new(5) MyClass[6];
  ...
}
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

delete

- **A `delete` *expression* invokes an object's destructor and then invokes `operator delete`**
  - `delete myPtr;`
- **The default `operator delete` returns memory to the heap**
- **Whenever `operator new` has been overloaded or overridden, you should also write `operator delete`**
  - If an object was not allocated from heap, the memory must not be returned to the heap



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Overriding `delete`

- **`operator delete` has one of the following signatures**
  - `void operator delete(void* address);`
  - `void operator delete(void* address, size_t size);`
    - Invoked only if the former function is not defined
  - The `delete` expression supplies the parameter values
- **This operator can be overridden for a specific class**

```
class MyClass{
  ...
  void operator delete(void* address){
    ...
  }
};
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Placement `delete`

- **Alternative signatures can be provided for `operator delete`**
  - Only invoked if processing an exception raised in a constructor called from a `new` expression involving a matching `operator new`
  - A matching operator is one where all parameters other than the first are of the same type; e.g.:

```
class MyClass{
  ...
  void* operator new(size_t size, void* address);

  MyClass(){
    throw std::exception();
  }

  void operator delete(void* address, void* arg2){
    // called if ctor from matching new throws exception
    ...
  }
};
```

```
MyClass* mc = new(0xFFFF) MyClass;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Deallocating Arrays

- **`operator delete` deallocates a single object**
- **To deallocate an array of objects, the separate `operator delete[]` is used**
  - `operator delete[]` must be overloaded or overridden explicitly
- **Which operators are called in the following code fragment?**

```
class MyClass{
  ...
  void* operator new[](size_t n, int x);
  void* operator new(size_t n);
  void operator delete[](void* a, int x);
  void operator delete[](void* a);
  void operator delete(void* a);
};
```

```
MyClass *ptr1 = new MyClass;
MyClass *ptr2 = new MyClass[4];
MyClass *ptr3 = new(5)MyClass[6];
delete ptr1;
delete[] ptr2;
delete[] ptr3;
```

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Memory Layout and Derived Classes



- **This section describes how a typical C++ compiler might arrange objects in memory**
- **This gives an insight into the trade-offs involved in different design choices**
  - Virtual vs. non-virtual functions
  - Single vs. multiple derivation
  - Virtual vs. non-virtual derivation

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Compiler Dependence



- **The C++ standard does not specify how a compiler should lay out objects or classes in memory**
  - It does specify behavior that must emerge from the compiler's strategy
  - This leads to certain approaches being favored
- **This section describes an approach to memory layout that is both compliant and commonly used**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Virtual Functions

- **Virtual functions are implemented using *virtual pointers* and *virtual tables***
  - Often abbreviated to *vptr* and *vtbl*
- **A virtual table is generated for any class that declares or inherits the declaration of virtual functions**
  - One table per class (not per object)
  - The table contains a pointer to the appropriate implementation of each virtual function applicable to the class
- **Each object of such a class contains a single virtual pointer, pointing to the virtual table**
- **This allows the compiler to generate code that will call the correct virtual function with minimal, fixed overhead**
  - One pointer per object
  - Two levels of indirection per call

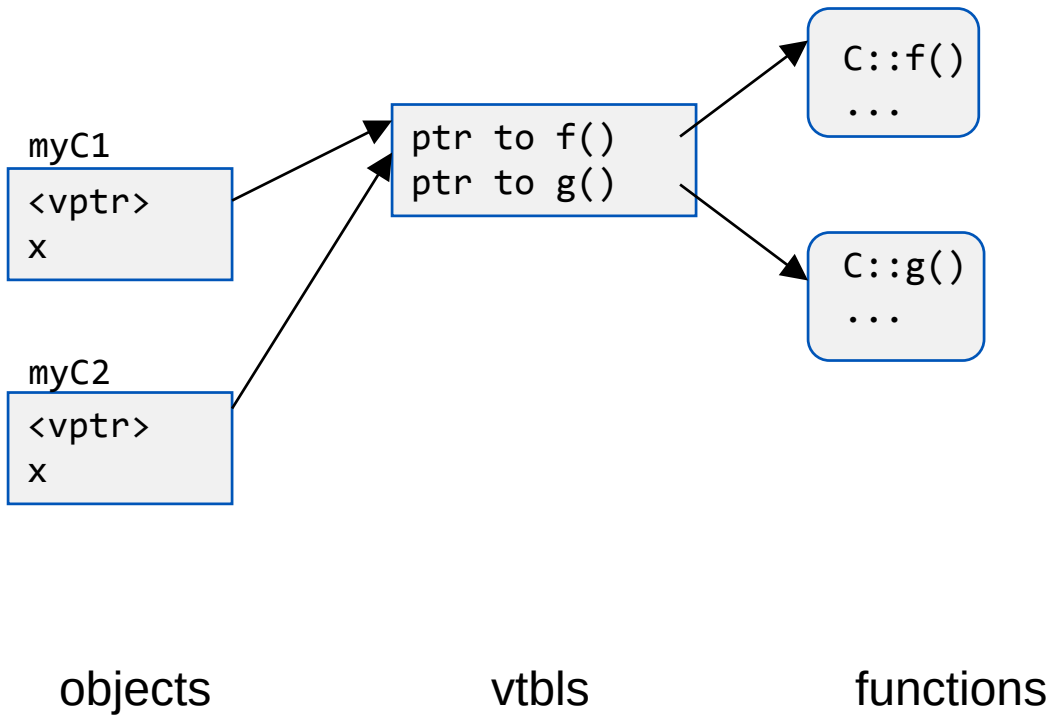
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Virtual Functions

```
class C {
public:
  int x;
  virtual void f();
  virtual void g();
};
```

```
C myC1;

C myC2;
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

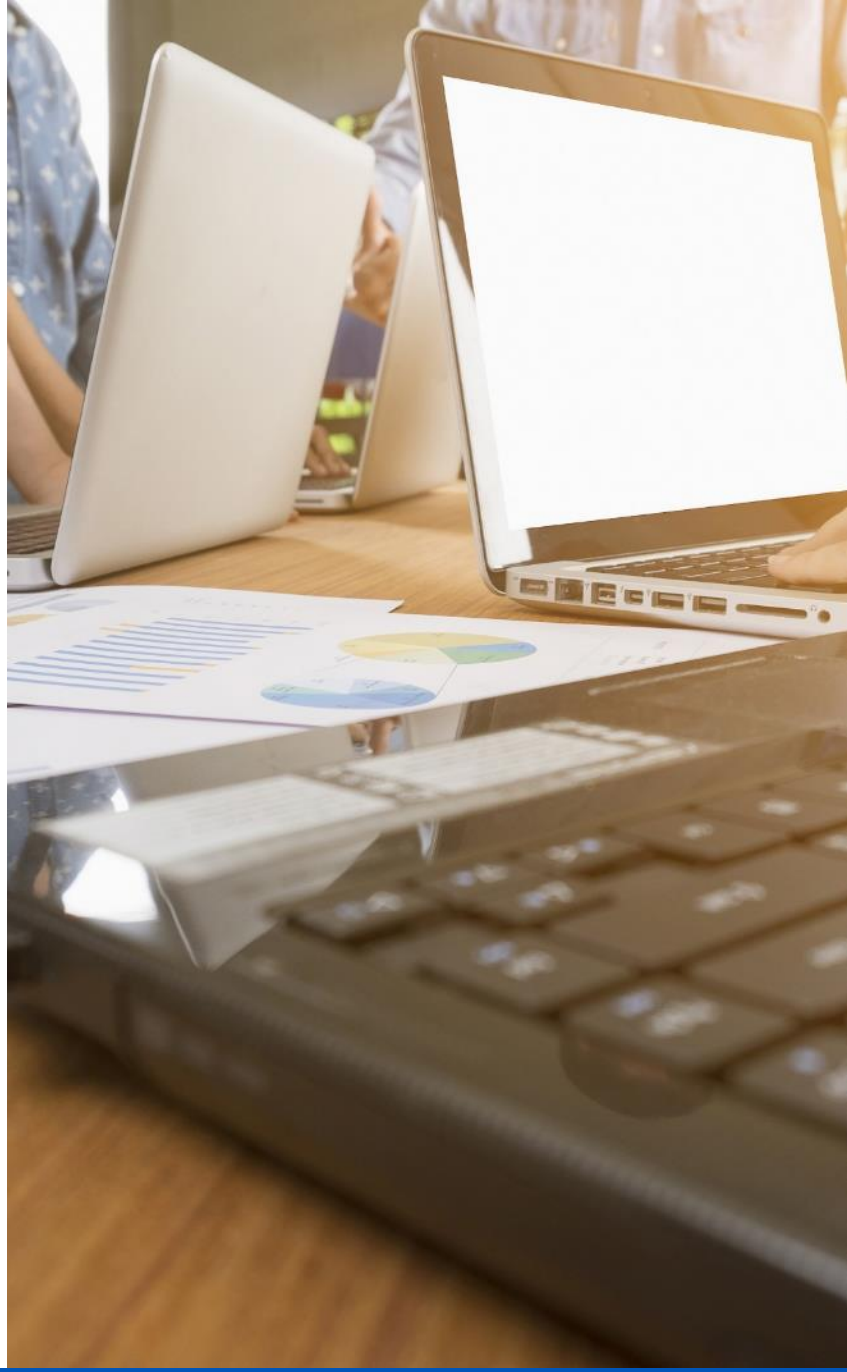
Contents

Memory Layout and Derivation

- Introduction

Single Derivation

- Multiple Derivation
- Virtual Derivation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Single Derivation

- **Objects of a singly derived class each contain a single virtual pointer**
- **The derived class has its own virtual table, which may point to different implementations than that of the base class**
- **To call a virtual function, the compiler generates the following code**
  - From the address of the object, follow the pointer to the virtual table
  - From the correct offset in the virtual table (e.g., `f()` is the first offset), follow the pointer to the code



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

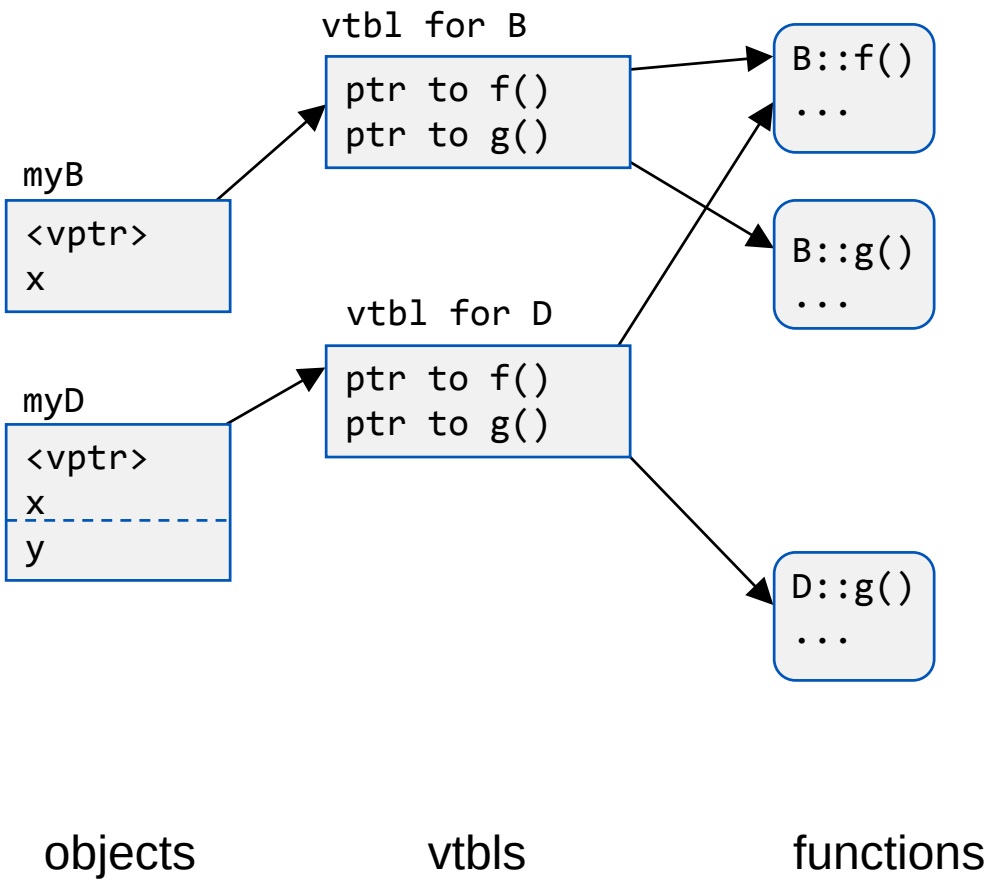
Single Derivation

```
class B {
  int x;
  virtual void f();
  virtual void g();
};

class D: public B {
  int y;
  void g();
};
```

```
B myB;

D myD;
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

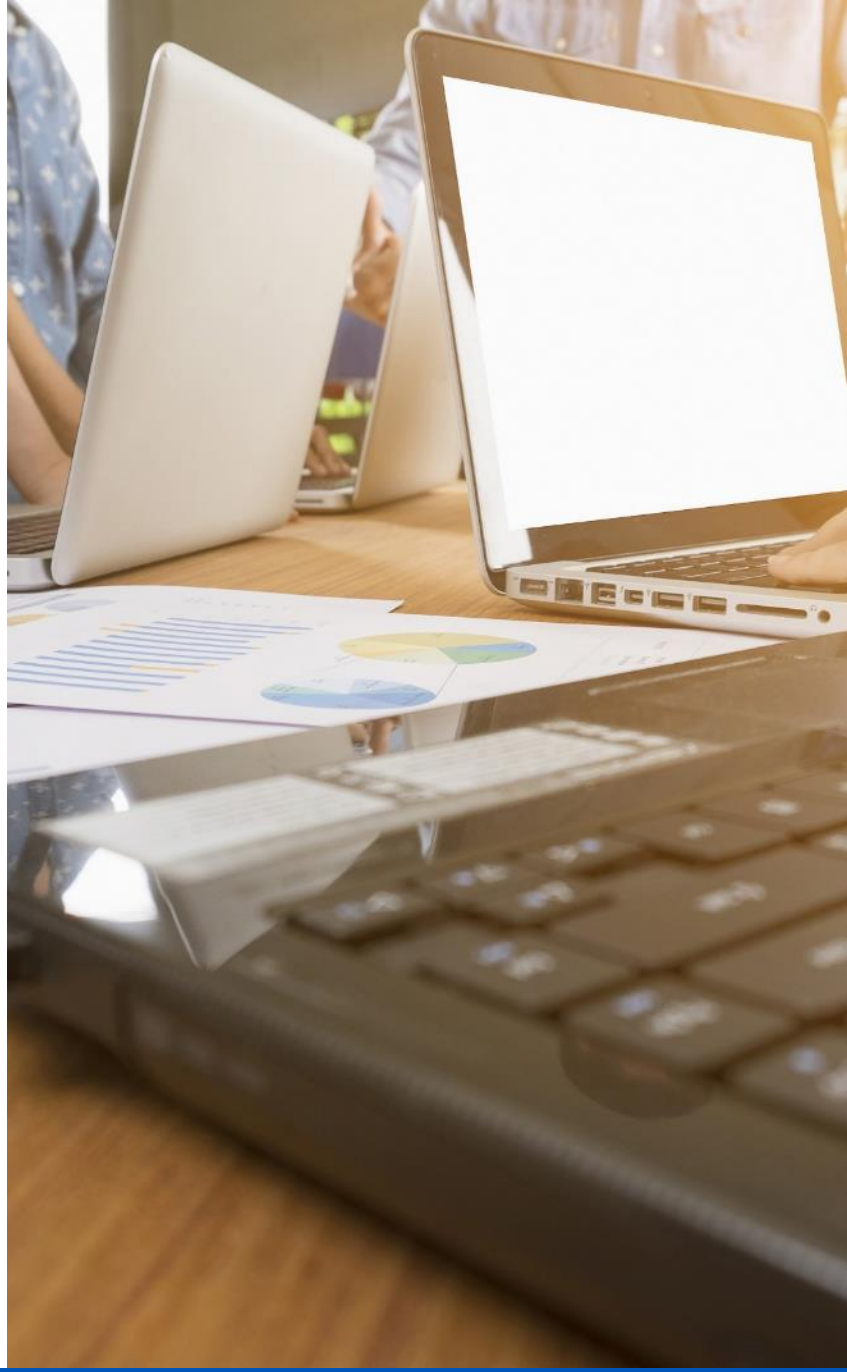
Contents

Memory Layout and Derivation

- Introduction
- Single Derivation

Multiple Derivation

- Virtual Derivation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Derivation

- **Objects of a multiply derived class will contain a complete copy of the memory layout for an object of each base class**
  - Multiple virtual pointers, each pointing to a different offset in the same virtual table
- **The value of `this` and of pointers to a multiply derived object may change if accessed through a base class other than the first**
  - This is to allow the code generation strategy to work under all circumstances
  - However, a virtual function call on a given object will always call the same definition, regardless of casting the object to one of its base classes

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Derivation

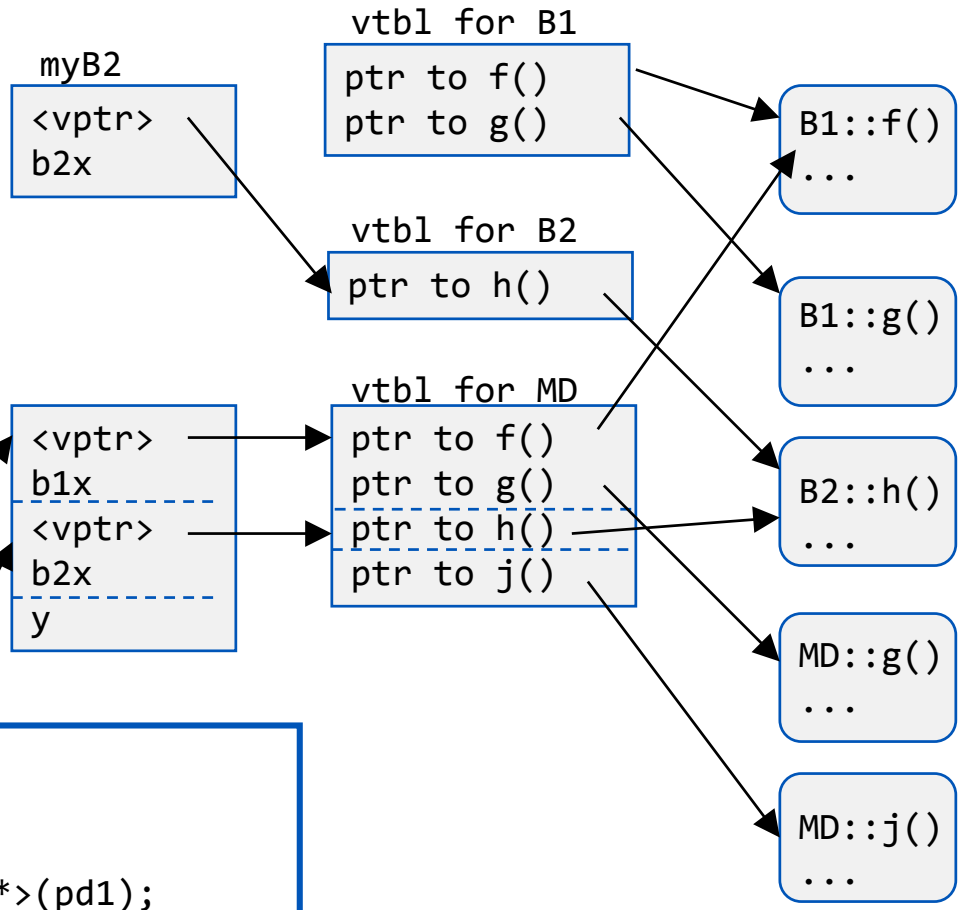
```
class B1 {
  int b1x;
  virtual void f();
  virtual void g();
};

class B2 {
  int b2x;
  virtual void h();
};

class MD: public B1,
          public B2 {
  int y;
  void g();
  virtual void j();
};
```

```
B2 myB2;

B1* pd1 = new MD;
B2* pd2 = dynamic_cast<B2*>(pd1);
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Common Ancestors

- **The preceding strategies are combined when a class is multiply derived from classes that share a common base class**
  - The derived object contains multiple copies of members declared in the common base class
  - The compiler must be able to determine a single, unambiguous definition for each virtual function call
  - Ambiguities must be resolved by the programmer using the scope operator

```
class CB {
  int cbx;
  virtual void f();
  virtual void g();
};
```

```
class CB1: public CB {
  int cb1x;
  void g();
};
```

```
class CB2: public CB {
  int cb2x;
  void f();
  virtual void h();
};
```

```
class CD: public CB1,
          public CB2 {
  int y;
  void g();
  virtual void j();
};
```

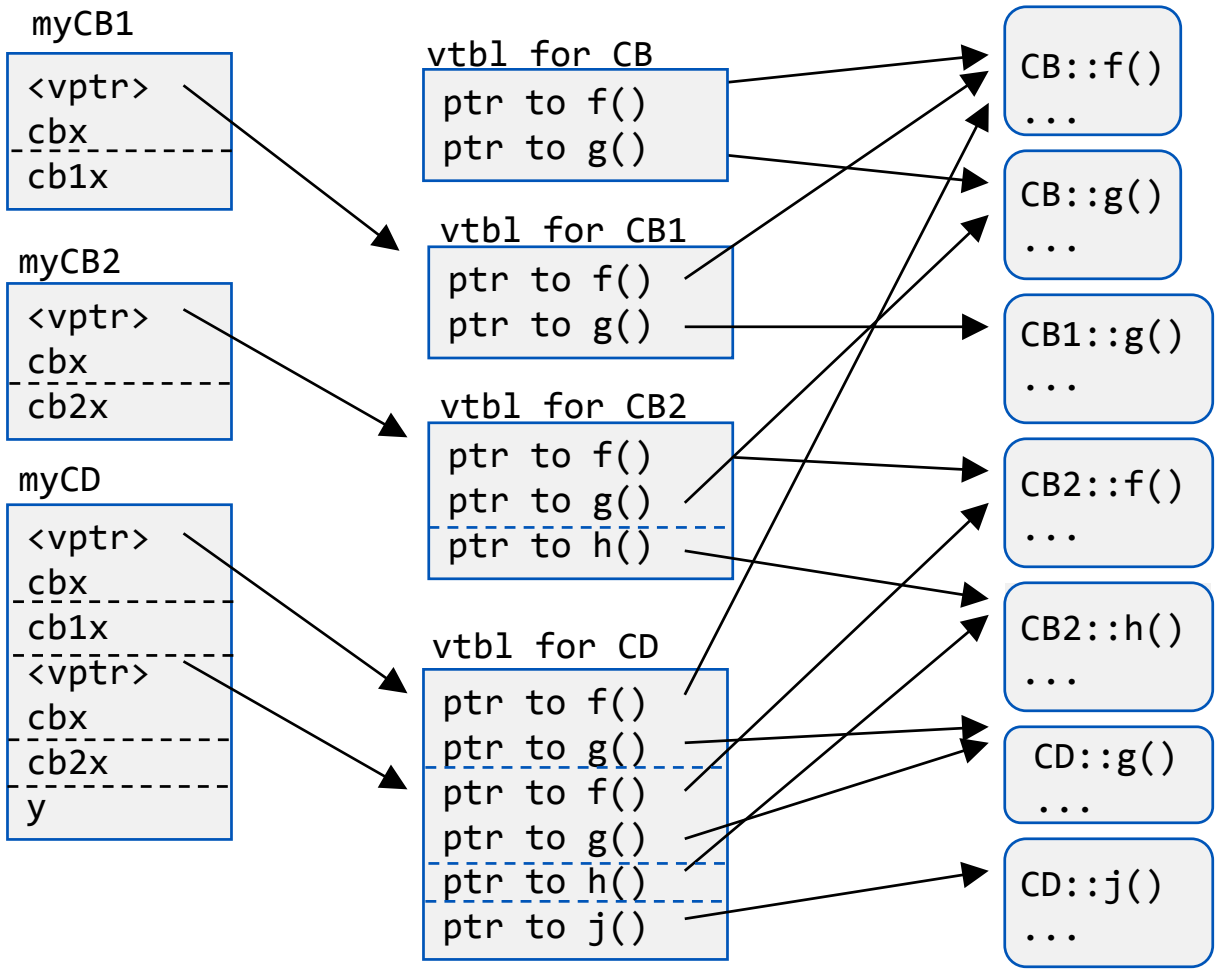
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Common Ancestors

```
CB1 myCB1;

CB2 myCB2;

CD myCD;
```



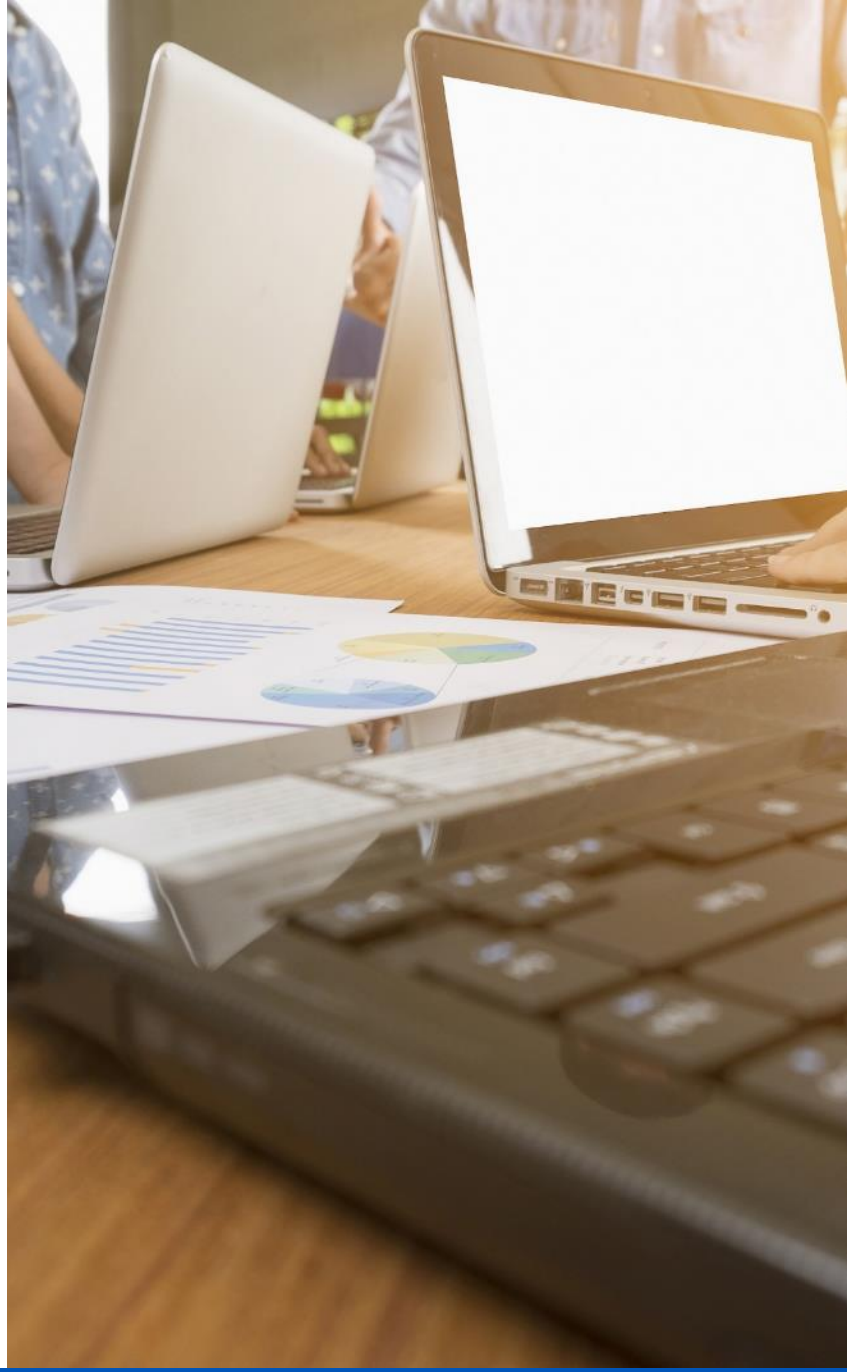
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Contents

Memory Layout and Derivation

- Introduction
- Single Derivation
- Multiple Derivation

Virtual Derivation



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Virtual Derivation

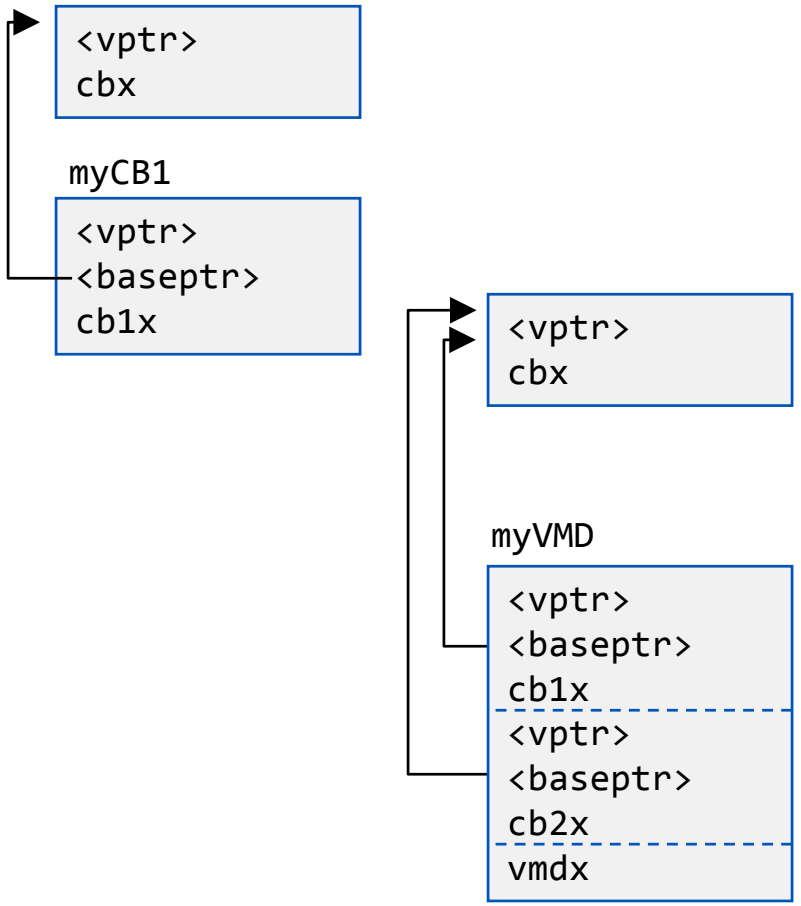
- **Using *virtual derivation* avoids inheriting multiple copies of common base class members**
  - This is more commonly what is intended
- **Virtual inheritance is used when deriving the intermediate classes**
  - Not the multiply derived child class
- **An object of a virtually derived class contains a pointer to its base class part, rather than including it directly**
  - This is to allow each intermediate part of a multiply derived child to point to the same area or storage
- **Java and C# do not support multiple inheritance. Why?**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Virtual Derivation

```
class CB {
  int cbx;
  virtual void f();
  virtual void g();
};
class CB1: virtual public CB {
  int cb1x;
  void g();
};
class CB2: virtual public CB {
  int cb2x;
  void f();
  virtual void h();
};
class VMD: public CB1, public CB2 {
  int vmdx;
};

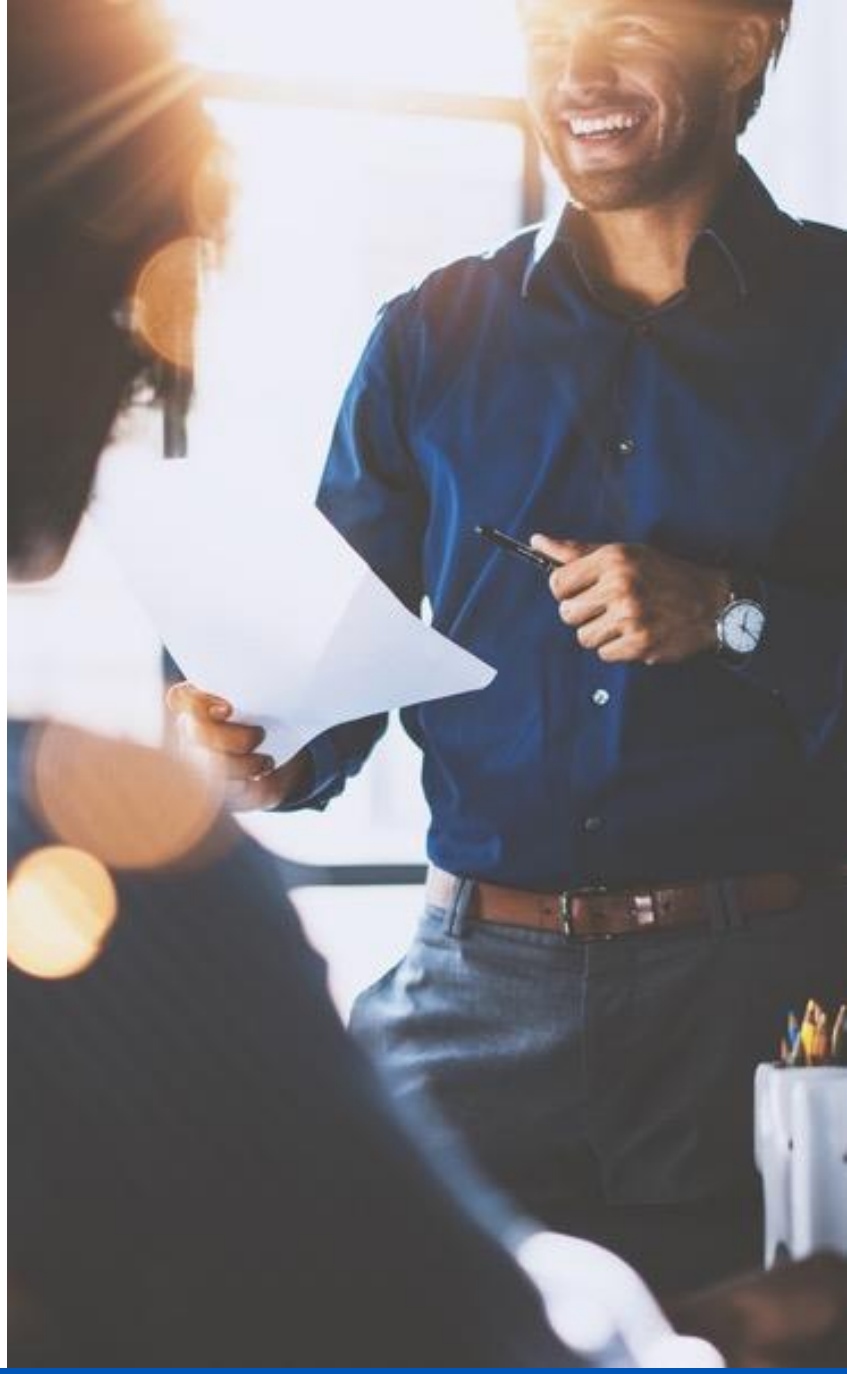
CB1 myCB1;
VMD myVMD;
```



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objectives

- **This appendix contains optional material that may be covered depending on interest and timing**
  - There will be no exam questions on this section
- **It expands two topics**
  - Overloading `new` and `delete` for detailed memory allocation handling
  - Memory layout issues and details of inheritance, virtual tables, multiple and virtual inheritance



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.