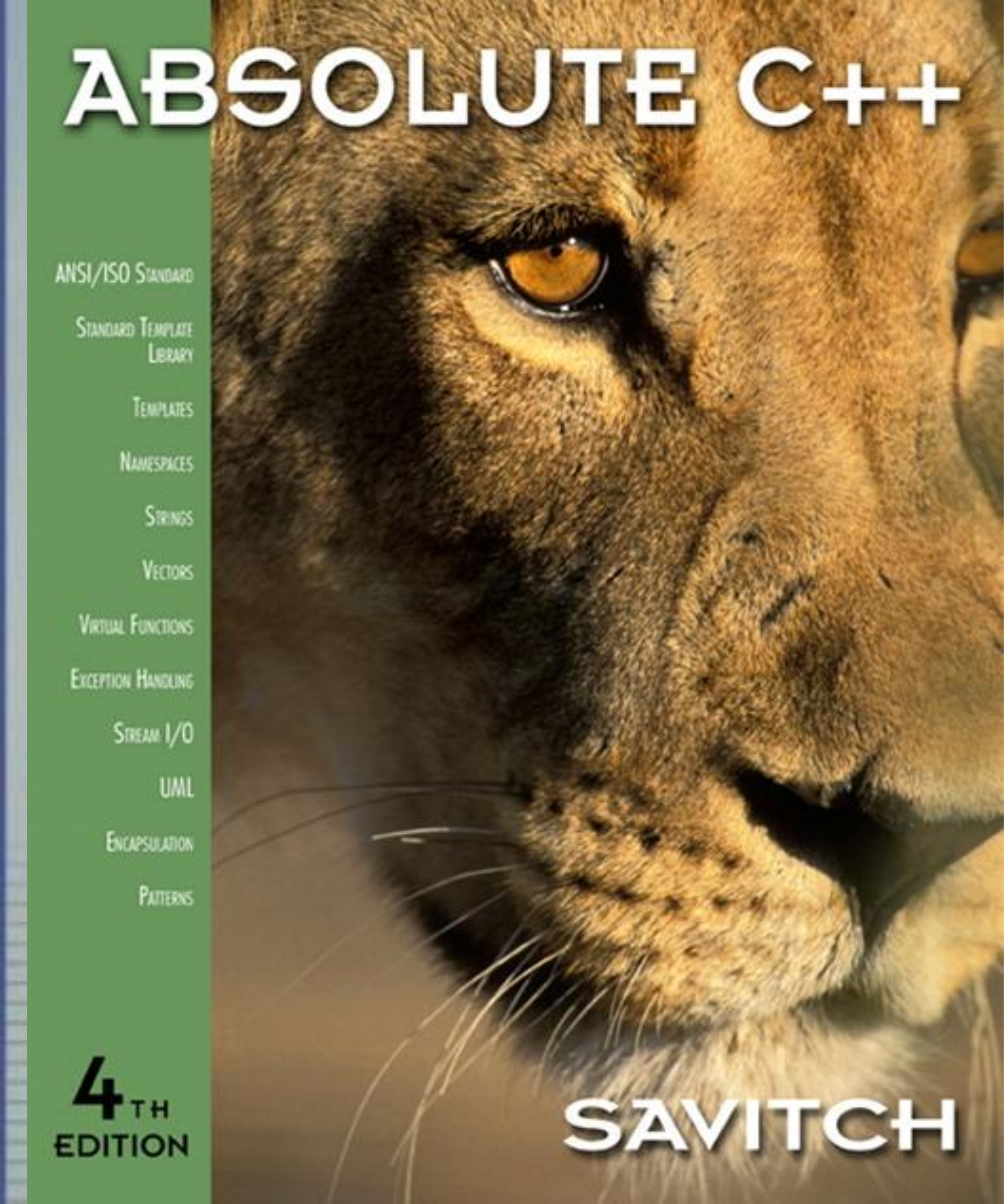
# CISC 181 final overview

- Next Tuesday, May 25—10:30 am-12:30 pm, Memorial 113
- Worth 20% of your grade
- Covers topics from class in chapters listed on course page from April 8 up to May 11 inclusive
  - Will <u>not</u> test on gdb, OpenGL (or recursion or exception handling)
- Format: Just like the midterm…only no tic-tac-toe programs ☺

# Topic list

| | |
|---|---|
| • Classes | Chap. 10.3, 14-14.1, 15-15.2 (through p. 678) |
| • Templates | Chap. 16-16.2 |
| • Linked data structures | Chap. 17-17.2 (through p. 763) |
| • STL | Chap. 7.3, 19-19.2 (through p. 872, + pp. 876-883) |




# Chapter 10

## Pointers, Dynamic Arrays, & **Classes**

# Chap. 10.3 Learning Objectives

- Classes, Pointers, Dynamic Arrays
  - The *this* pointer
  - Destructors, copy constructors

# Back to Classes

- The -> operator
  - Shorthand notation
- Combines dereference operator, *, and dot operator
- Specifies member of class "pointed to" by given pointer
- Example:
  MyClass *p;
  p = new MyClass;
  p->grade = "A";   Equivalent to:
  (*p).grade = "A";

# The this Pointer

- Member function definitions might need to refer to calling object
- Use predefined *this* pointer
  - Automatically points to calling object:

```
Class Simple
{
public:
  void showStuff() const;
private:
  int stuff;
};
```

- Two ways for member functions to access:

```
cout << stuff;
cout << this->stuff;
```

# Overloading Assignment Operator

- Assignment operator returns reference
  - So assignment "chains" are possible
  - e.g., a = b = c;
    - Sets a and b equal to c
- Operator must return "same type" as its left-hand side
  - To allow chains to work
  - The *this* pointer will help with this!

# Overloading Assignment Operator

- Recall: Assignment operator must be member of the class
  - It has one parameter
  - Left-operand is calling object
    s1 = s2;
    - Think of like: s1.=(s2);
- s1 = s2 = s3;
  - Requires (s1 = s2) = s3;
  - So (s1 = s2) must return object of s1"s type
    - And pass to " = s3";

# Overloaded = Operator Definition

- Uses string Class example:

```
StringClass& StringClass::operator=(const StringClass& rtSide)
{
	if (this == &rtSide)		// if right side same as left side
			return *this;
	else
	{
			capacity = rtSide.length;
			length
			length = rtSide.length;
			delete [] a;
			a = new char[capacity];
			for (int I = 0; I < length; I++)
				a[I] = rtSide.a[I];
			return *this;
	}
}
```

# Shallow and Deep Copies

- Shallow copy
  - Assignment copies only member variable contents over
  - Default assignment and copy constructors
- Deep copy
  - Pointers, dynamic memory involved
  - Must dereference pointer variables to "get to" data for copying
  - Write your own assignment overload and copy constructor in this case!

# Destructor Need

- Dynamically-allocated variables
  - Do not go away until "deleted"
- If pointers are only private member data
  - They dynamically allocate "real" data
    - In constructor
  - Must have means to "deallocate" when object is destroyed
- Answer: destructor!

# Destructors

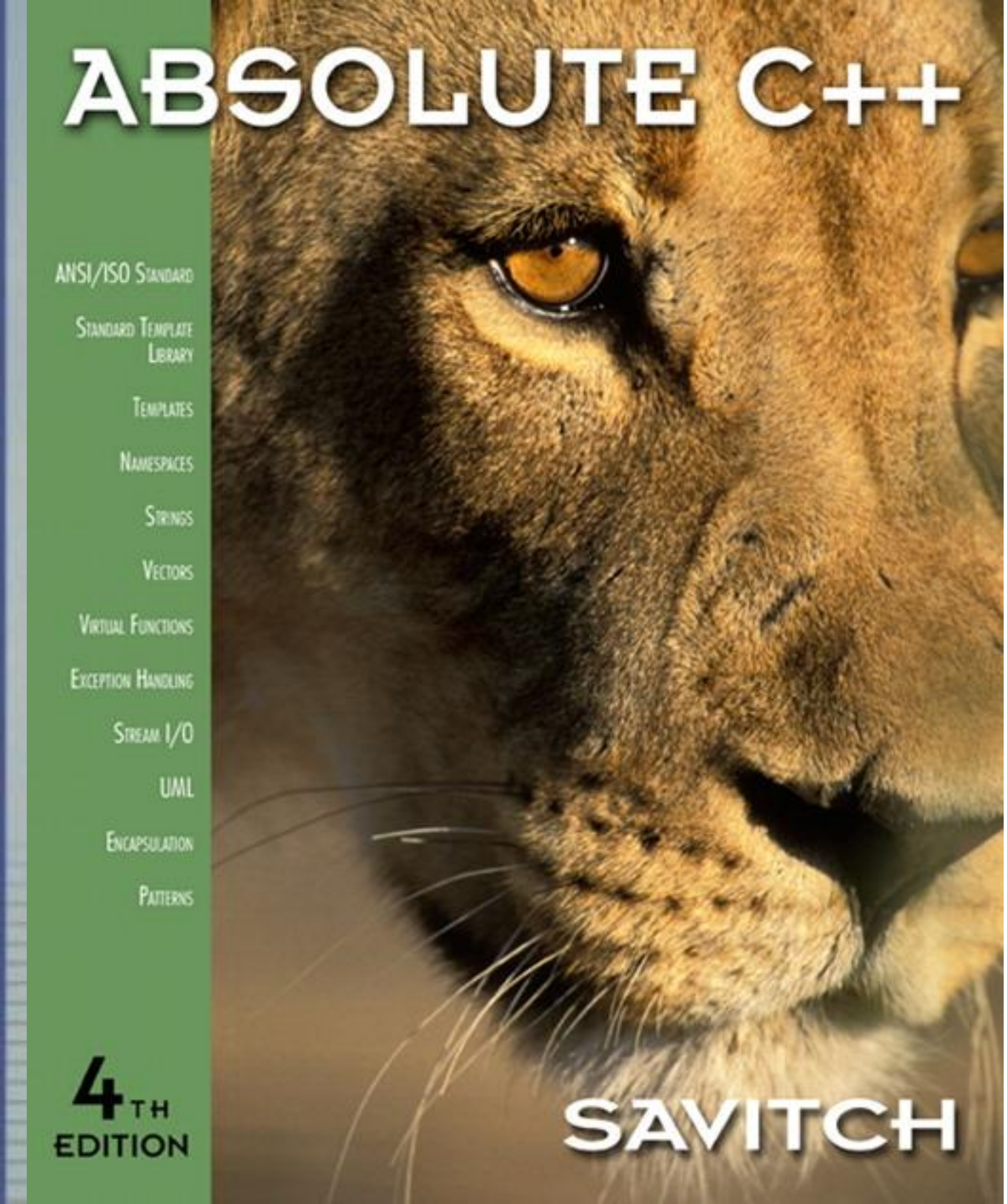
- Opposite of constructor
  - Automatically called when object is out-of-scope
  - Default version only removes ordinary variables, not dynamic variables
- Defined like constructor, just add ~
  - ```
    MyClass::~MyClass()
    {
      //Perform delete clean-up duties
    }
    ```
    ```

# Copy Constructors

- Automatically called when:
  1. Class object declared and initialized to other object
  2. When function returns class type object
  3. When argument of class type is "plugged in"
     as actual argument to call-by-value parameter
- Requires "temporary copy" of object
  – Copy constructor creates it
- Default copy constructor
  – Like default "=", performs member-wise copy
- Pointers → write own copy constructor!

# The "Big 3"

- If you do one, you'll probably need to do all because of new/delete issue
  1. Overloading assignment operator
  2. Copy constructor
  3. Destructor
- Also, these are not inherited (nor is constructor)




# Chapter 14

## Inheritance

# Learning Objectives

- Inheritance Basics
  - Derived classes, with constructors
  - Redefining member functions
  - Non-inherited functions

# Introduction to Inheritance

- Object-oriented programming
  - Powerful programming technique
  - Provides abstraction dimension called *inheritance*
- General form of class is defined
  - Specialized versions then inherit properties of general class
  - And add to it/modify its functionality for its appropriate use

# Inheritance Basics

- New class inherited from another class
- Base class
  - "General" class from which others derive
- Derived class
  - New class
  - Automatically has base class's:
    - Member variables
    - Member functions
  - Can then add additional member functions and variables

# Derived Classes

- Consider example:
  Class of "Employees"
- Composed of:
  - Salaried employees
  - Hourly employees
- Each is "subset" of employees
  - Another might be those paid fixed wage each month or week

# Derived Classes

- Don’t "need" type of generic "employee"
  - Since no one’s just an "employee"
- General concept of employee helpful!
  - All have names
  - All have social security numbers
  - Associated functions for these "basics" are same among all employees
- So "general" class can contain all these "things" about employees

# Employee Class

- Many members of "employee" class apply to all types of employees
  - Accessor functions
  - Mutator functions
  - Most data items:
    - SSN
    - Name
    - Pay
- We won’t have "objects" of this class, however

# Employee Class

- Consider printCheck() function:
  - Will always be "redefined" in derived classes
  - So different employee types can have different checks
  - Makes no sense really for "undifferentiated" employee
  - So function printCheck() in Employee class says just that
    - Error message stating "printCheck called for undifferentiated employee!! Aborting…"

# Deriving from Employee Class

- Derived classes from Employee class:
  - Automatically have all member variables
  - Automatically have all member functions
- Derived class said to "inherit" members from base class
- Can then redefine existing members and/or add new members

# **Display 14.3** Interface for the Derived Class HourlyEmployee (1 of 2)

**Display 14.3 Interface for the Derived Class** HourlyEmployee

```
//This is the header file hourlyemployee.h.
//This is the interface for the class HourlyEmployee.
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H

#include <string>
#include "employee.h"

using std::string;

namespace SavitchEmployees
{
```

# **Display 14.3** Interface for the Derived Class HourlyEmployee (2 of 2)

```
    class HourlyEmployee : public Employee
    {
    public:
        HourlyEmployee( );
        HourlyEmployee(string theName, string theSsn,
                                   double theWageRate, double theHours);
        void setRate(double newWageRate);
        double getRate( ) const;
        void setHours(double hoursWorked);
        double getHours( ) const;
        void printCheck( ) ;
    private:
        double wageRate;
        double hours;
    };

}//SavitchEmployees

#endif //HOURLYEMPLOYEE_H
```

*You only list the declaration of an inherited member function if you want to change the definition of the function.*

# HourlyEmployee Class Interface

- Note definition begins same as any other
  - #ifndef structure
  - Includes required libraries
  - Also includes employee.h!
- And, the heading:
  class HourlyEmployee : public Employee
  { …
  - Specifies "publicly inherited" from Employee class

# HourlyEmployee Class Additions

- Derived class interface only lists new or "to be redefined" members
  - Since all others inherited are already defined
  - i.e.: "all" employees have ssn, name, etc.
- HourlyEmployee adds:
  - Constructors
  - wageRate, hours member variables
  - setRate(), getRate(), setHours(), getHours() member functions

# HourlyEmployee Class Redefinitions

- HourlyEmployee redefines:
  - printCheck() member function
  - This "overrides" the printCheck() function implementation from Employee class
- Its definition must be in HourlyEmployee class's implementation
  - As do other member functions declared in HourlyEmployee's interface
    - New and "to be redefined"

# Inheritance Terminology

- Common to simulate family relationships
- Parent class
  - Refers to base class
- Child class
  - Refers to derived class
- Ancestor class
  - Class that's a parent of a parent …
- Descendant class
  - Opposite of ancestor

# Constructors in Derived Classes

- Base class constructors are NOT inherited in derived classes!
  - But they can be invoked within derived class constructor
    - Which is all we need!
- Base class constructor must initialize all base class member variables
  - Those inherited by derived class
  - So derived class constructor simply calls it
    - "First" thing derived class constructor does

# Derived Class Constructor Example

- Consider syntax for HourlyEmployee constructor:

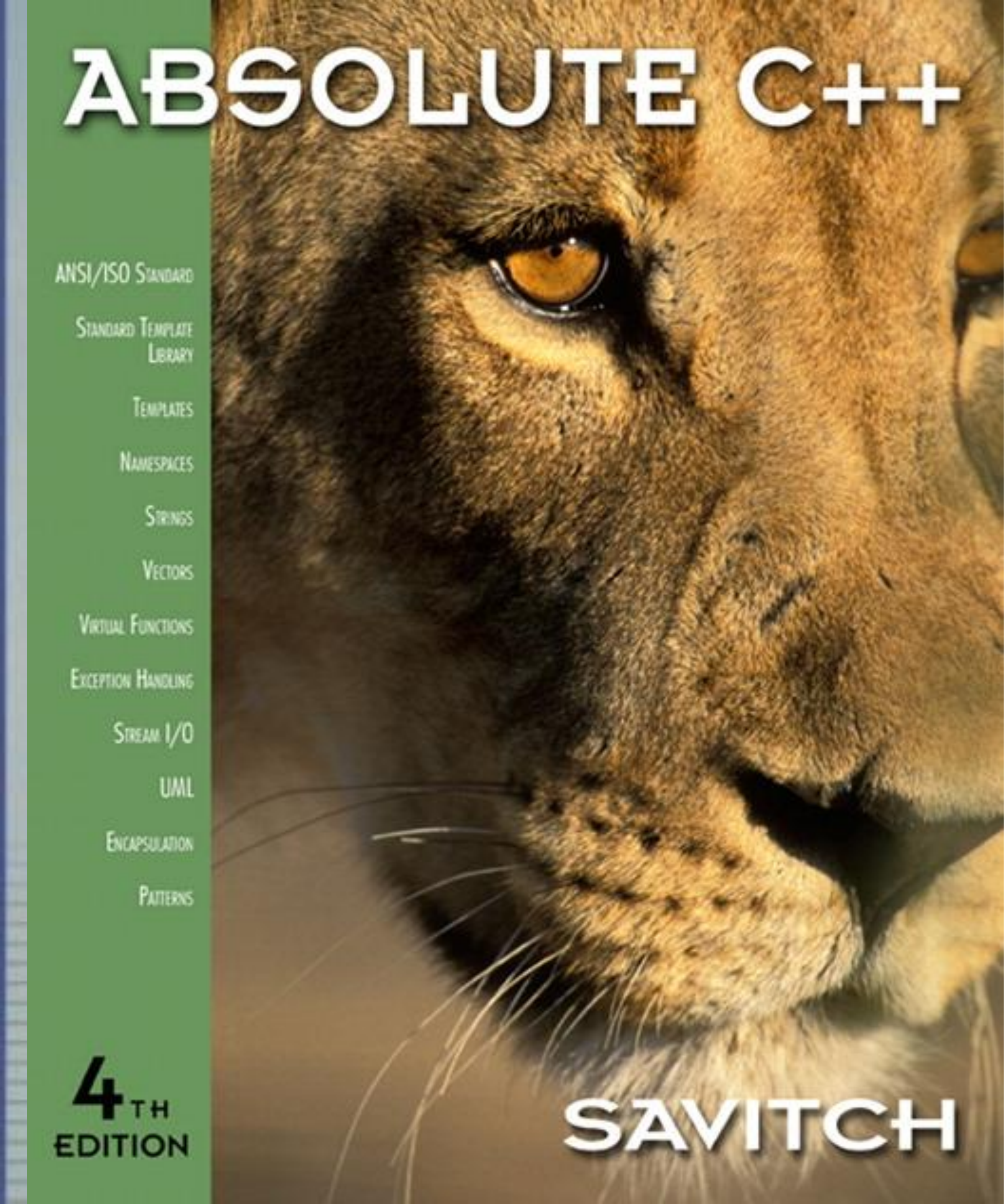
```
HourlyEmployee::HourlyEmployee(string theName,
                        string theNumber, double theWageRate,
                        double theHours)
                : Employee(theName, theNumber),
                wageRate(theWageRate), hours(theHours)
{
        //Deliberately empty
}
```

- Portion after : is "initialization section"
  - Includes invocation of Employee constructor

# Constructor: No Base Class Call

- Derived class constructor should always invoke one of the base class's constructors
- If you do not:
  - Default base class constructor automatically called
- Equivalent constructor definition:

```
HourlyEmployee::HourlyEmployee()
                            : wageRate(0), hours(0)
{ }
```




# Chapter 15

## Polymorphism and Virtual Functions

# Learning Objectives

- Virtual Function Basics
  - Late binding
  - Implementing virtual functions
  - When to use a virtual function
  - Abstract classes and pure virtual functions
- Pointers and Virtual Functions
  - Extended type compatibility
  - Downcasting and upcasting

# Virtual Function Basics

- Polymorphism
  - Associating many meanings to one function
  - Virtual functions provide this capability
  - Fundamental principle of object-oriented programming!
- Virtual
  - Existing in "essence" though not in fact
- Virtual Function
  - Can be "used" before it’s "defined"

# Figures Example

- Best explained by example:
- Classes for several kinds of figures
  - Rectangles, circles, ovals, etc.
  - Each figure an object of different class
    - Rectangle data: height, width, center point
    - Circle data: center point, radius
- All derive from one parent-class: Figure
- Require function: draw()
  - Different instructions for each figure

# Figures Example 2

- Each class needs different *draw* function
- Can be called "draw" in each class, so:

```
Rectangle r;
Circle c;
r.draw();  //Calls Rectangle class's draw
c.draw(); //Calls Circle class's draw
```

- Nothing new here yet…

# Figures Example: center()

- Parent class Figure contains functions that apply to "all" figures; consider: center(): moves a figure to center of screen
  - Erases 1st, then re-draws
  - So Figure::center() would use function draw() to re-draw
  - Complications!
    - Which draw() function?
    - From which class?

# Figures Example: New Figure

- Consider new kind of figure comes along:
  Triangle class
  derived from Figure class
- Function center() inherited from Figure
  - Will it work for triangles?
  - It uses draw(), which is different for each figure!
  - It will use Figure::draw() → won't work for triangles
- Want inherited function center() to use function Triangle::draw() NOT function Figure::draw()
  - But class Triangle wasn't even WRITTEN when Figure::center() was!  Doesn't know "triangles"!

# Figures Example: Virtual!

- Virtual functions are the answer
- Tells compiler:
  - "Don’t know how function is implemented"
  - "Wait until used in program"
  - "Then get implementation from object instance"
- Called late binding or dynamic binding
  - Virtual functions implement late binding

# Virtual: How?

- To write C++ programs:
  - Assume it happens by "magic"!
- But explanation involves late binding
  - Virtual functions implement late binding
  - Tells compiler to "wait" until function is used in program
  - Decide which definition to use based on calling object
- Very important OOP principle!

# Overriding

- Virtual function definition changed in a derived class
  - We say it's been "overidden"
- Similar to redefined
  - Recall: for standard functions
- So:
  - Virtual functions changed: ***overridden***
  - Non-virtual functions changed: ***redefined***

# Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we've seen
- One major disadvantage: overhead!
  - Uses more storage
  - Late binding is "on the fly", so programs run slower
- So if virtual functions not needed, should not be used

# Pure Virtual Functions

- Base class might not have "meaningful" definition for some of its members!
  - Its purpose is solely for others to derive from
- Recall class Figure
  - All figures are objects of derived classes
    - Rectangles, circles, triangles, etc.
  - Class Figure has no idea how to draw!
- Make it a pure virtual function:
  virtual void draw() = 0;

# Abstract Base Classes

- Pure virtual functions require no definition
  - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is: abstract base class
  - Can only be used as base class
  - No objects can ever be created from it
    - Since it doesn’t have complete "definitions" of all its members!
- If derived class fails to define all pure’s:
  - It’s an abstract base class too

# Extended Type Compatibility

- Given:
  Derived is derived class of Base
  - Derived objects can be assigned to objects of type Base
  - But NOT the other way!
- Consider next example:
  - A Dog "is a" Pet, but reverse not true

# Extended Type Compatibility Example

- ```
  class Pet
  {
  public:
          string name;
          virtual void print() const;
  };
  class Dog : public Pet
  {
  public:
          string breed;
          virtual void print() const;
  };
  ```
  ```

# Classes Pet and Dog

- Now given declarations:
  Dog vdog;
  Pet vpet;
- Notice member variables name and breed are public!
  - For example purposes only!  Not typical!

# Using Classes Pet and Dog

- Anything that "is a" dog "is a" pet:
  - vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;
  - These are allowable
- Can assign values to parent-types, but not reverse
  - vdog = vpet not allowed

# Slicing Problem

- Notice value assigned to vpet "loses" its breed field!
  - cout << vpet.breed;
    - Produces ERROR msg!
  - Called slicing problem
- Might seem appropriate
  - Dog was moved to Pet variable, so it should be treated like a Pet
    - And therefore not have "dog" properties
  - Makes for interesting philosphical debate

# Slicing Problem Fix

- In C++, slicing problem is nuisance
  - It still "is a" Great Dane named Tiny
  - We'd like to refer to its breed even if it's been treated as a Pet
- Can do so with pointers to dynamic variables

# Slicing Problem Example

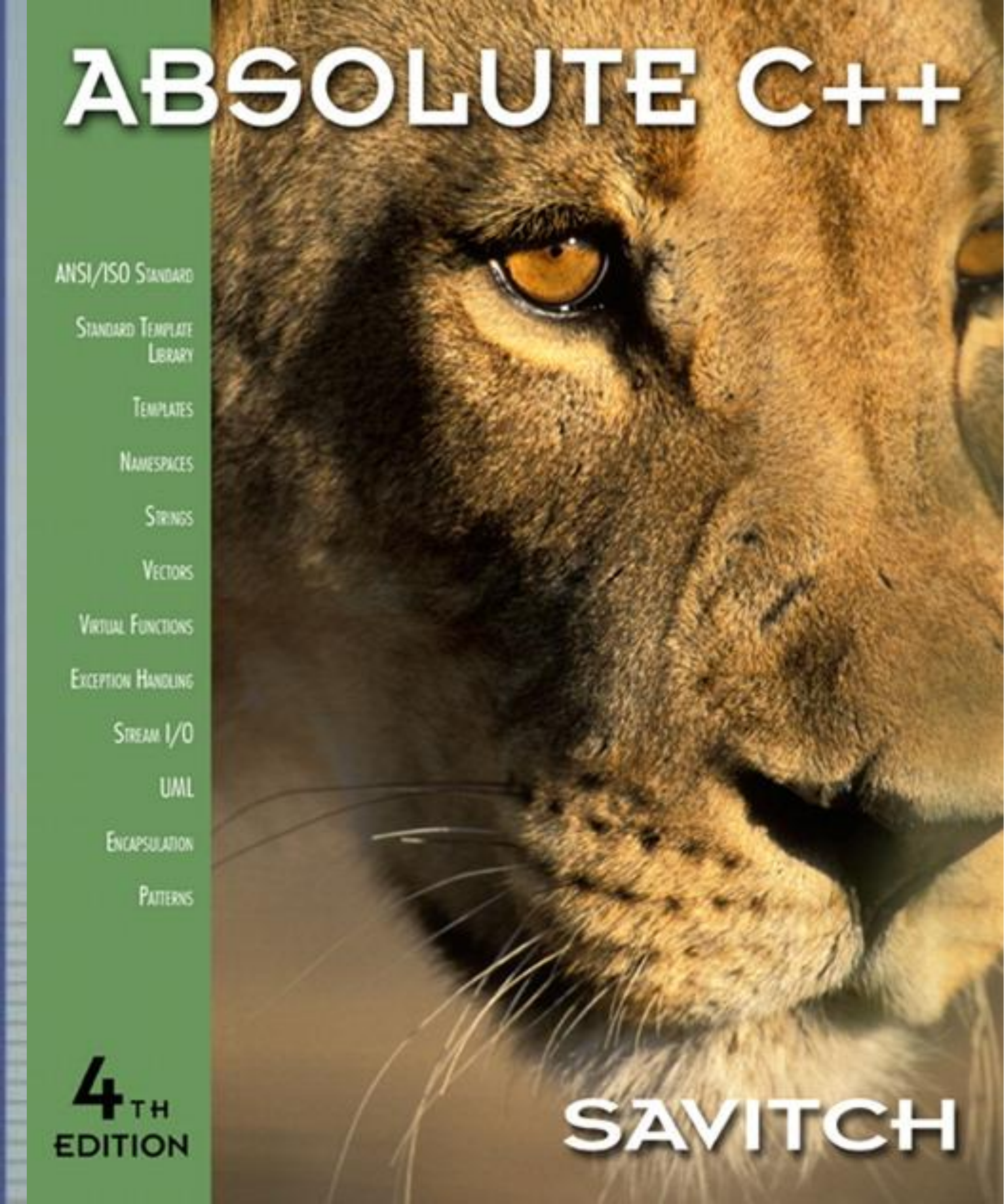
- Pet *ppet;
  Dog *pdog;
  pdog = new Dog;
  pdog->name = "Tiny";
  pdog->breed = "Great Dane";
  ppet = pdog;
- Cannot access breed field of object pointed to by ppet:
  cout << ppet->breed;        //ILLEGAL!

# Slicing Problem Example

- Must use virtual member function:
  ppet->print();
  - Calls print member function in Dog class!
    - Because it's virtual
  - C++ "waits" to see what object pointer ppet is actually pointing to before "binding" call

# Virtual Destructors

- Recall: destructors needed to de-allocate dynamically allocated data
- Consider:
  ```
  Base *pBase = new Derived;
  …
  delete pBase;
  ```
  - Would call base class destructor even though pointing to Derived class object!
  - Making destructor ***virtual*** fixes this!
- Good policy for all destructors to be virtual




# Chapter 16

## Templates

# Learning Objectives

- Function Templates
  - Syntax, defining
  - Compiler complications
- Class Templates
  - Syntax
  - Example: array template class

# Introduction

- C++ templates
  - Allow very "general" definitions for functions and classes
  - Type names are "parameters" instead of actual types
  - Precise definition determined at run-time

# Function Templates

- Recall function swapValues:

```
void swapValues(int& var1, int& var2)
{
      int temp;
      temp = var1;
      var1 = var2;
      var2 = temp;
}
```

- Applies only to variables of type int
- But code would work for any types!

# Function Templates vs. Overloading

- Could overload function for char's:

```
void swapValues(char& var1, char& var2)
{
     char temp;
     temp = var1;
     var1 = var2;
     var2 = temp;
}
```

- But notice: code is nearly identical!
  - Only difference is type used in 3 places

# Function Template Syntax

- Allow "swap values" of any type variables:

```
template<class T>   // I use typename instead of class
void swapValues(T& var1, T& var2)
{
    T temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

- First line called "template prefix"
  - Tells compiler what's coming is "template"
  - And that T is a type parameter

# Template Prefix

- Recall:
  template<class T>
- In this usage, "class" means "type", or "classification"
- Can be confused with other "known" use of word "class"!
  - C++ allows keyword "typename" in place of keyword "class" here
  - But most use "class" anyway

# Template Prefix 2

- Again:
  template<class T>
- T can be replaced by any type
  - Predefined or user-defined (like a C++ class type)
- In function definition body:
  - T used like any other type
- Note: can use other than "T", but T is "traditional" usage

# Function Template Definition

- swapValues() function template is actually large "collection" of definitions!
  – A definition for each possible type!
- Compiler only generates definitions when required
  – But it's "as if" you'd defined for all types
- Write one definition → works for all types that might be needed

# Compiler Complications

- Function declarations and definitions
  - Typically we have them separate
  - For templates → not supported on most compilers!
- Safest to place template function definition in file where invoked
  - Many compilers require it appear 1st
  - Often we #include all template definitions

# More Compiler Complications

- Check your compiler's specific requirements
  - Some need to set special options
  - Some require special order of arrangement of template definitions vs. other file items
- Most usable template program layout:
  - Template definition in same file it's used
  - Ensure template definition precedes all uses
    - Can #include it

# Multiple Type Parameters

- Can have:
  template<class T1, class T2>
- Not typical (but gets used for STL maps…)
  - Usually only need one "replaceable" type
  - Cannot have "unused" template parameters
    - Each must be "used" in definition
    - Error otherwise!

# Algorithm Abstraction

- Refers to implementing templates
- Express algorithms in "general" way:
  - Algorithm applies to variables of any type
  - Ignore incidental detail
  - Concentrate on substantive parts of algorithm
- Function templates are one way C++ supports algorithm abstraction

# Defining Templates Strategies

- Develop function normally
  - Using actual data types
- Completely debug "ordinary" function
- Then convert to template
  - Replace type names with type parameter as needed
- Advantages:
  - Easier to solve "concrete" case
  - Deal with algorithm, not template syntax

# Inappropriate Types in Templates

- Can use any type in template for which code makes "sense"
  - Code must behave in appropriate way
- e.g., swapValues() template function
  - Cannot use type for which assignment operator isn't defined
  - Example: an array:
    int a[10], b[10];
    swapValues(a, b);
    - Arrays cannot be "assigned"!

# Class Templates

- Can also "generalize" classes
  template<class T>
  - Can also apply to class definition
  - All instances of "T" in class definition replaced by type parameter
  - Just like for function templates!
- Once template defined, can declare objects of the class

# Class Template Definition

- ```
  template<class T>
  class Pair
  {
  public:
          Pair();
          Pair(T firstVal, T secondVal);
          void setFirst(T newVal);
          void setSecond(T newVal);
          T getFirst() const;
          T getSecond() const;
  private:
          T first; T second;
  };
  ```
  ```

# Class Templates as Parameters

- Consider:
  int addUP(const Pair<int>& the Pair);
  - The type (int) is supplied to be used for T in defining this class type parameter
  - It "happens" to be call-by-reference here
- Again: template types can be used anywhere standard types can

# Class Templates Within Function Templates

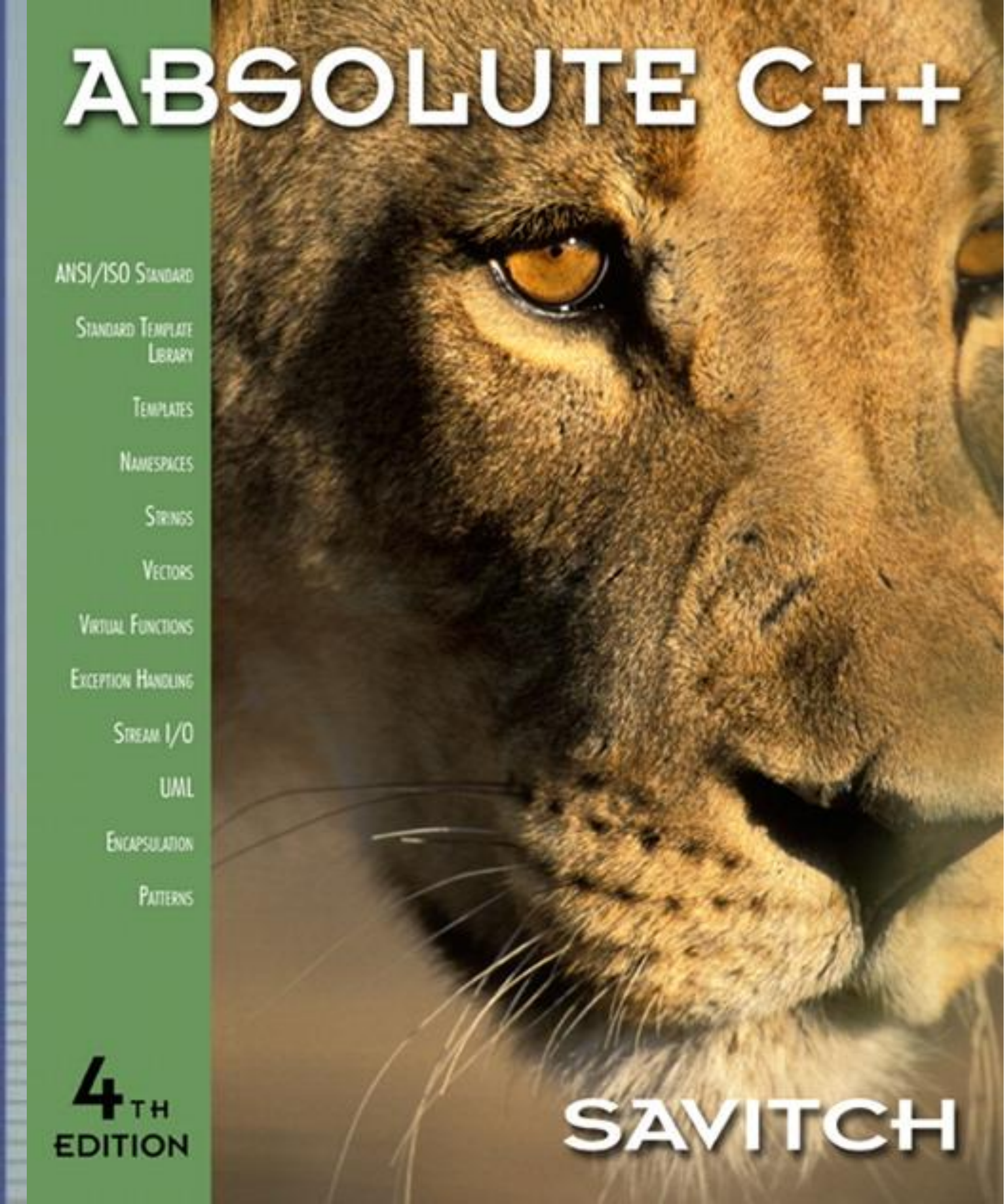
- Rather than defining new overload:

```
template<class T>
T addUp(const Pair<T>& the Pair);
//Precondition: Operator + is defined for values
                    of type T
//Returns sum of two values in thePair
```

- Function now applies to all kinds of numbers

# Restrictions on Type Parameter

- Only "reasonable" types can be substituted for T
- Consider:
  - Assignment operator must be "well-behaved"
  - Copy constructor must also work
  - If T involves pointers, then destructor must be suitable!
- Similar issues as function templates




# Chapter 17

## Linked Data Structures

# Learning Objectives

- Nodes and Linked Lists
  - Creating, searching
- Linked List Applications
  - Stacks

# Introduction

- Linked list
  - Constructed using pointers
  - Grows and shrinks during run-time
  - Doubly Linked List : A variation with pointers in both directions
- Pointers backbone of such structures
  - Use dynamic variables
- Standard Template Library
  - Has predefined versions of some structures

# Approaches

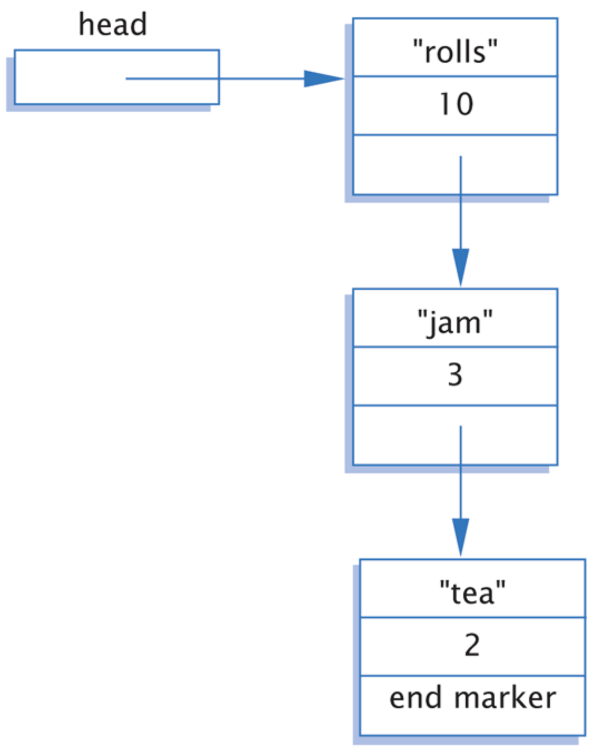
- Three ways to handle such data structures:
  1. C-style approach: global functions and structs with everything public
  2. Classes with private member variables and accessor and mutator functions
  3. Friend classes

# Nodes and Linked Lists

- Linked list
  - Simple example of "dynamic data structure"
  - Composed of nodes
- Each "node" is variable of struct or class type that's dynamically created with new
  - Nodes also contain pointers to other nodes
  - Provide "links"

# **Display 17.1** Nodes and Pointers

**Display 17.1 Nodes and Pointers**



# Node Definition

- struct ListNode
  {
  	string item;
  	int count;
  	ListNode *link;
  };
  typedef ListNode* ListNodePtr;
- Order here is important!
  - Listnode defined 1st, since used in typedef
- Also notice "circularity"

# Head Pointer

- Box labeled "head" not a node:
  ListNodePtr head;
  - A simple pointer to a node
  - Set to point to 1st node in list
- Head used to "maintain" start of list
- Also used as argument to functions

# Example Node Access

- (*head).count = 12;
  - Sets *count* member of node pointed to by *head* equal to 12
- Alternate operator, ->
  - Called "arrow operator"
  - Shorthand notation that combines * and .
  - head->count = 12;
    - Identical to above
- cin >> head->item
  - Assigns entered string to *item* member

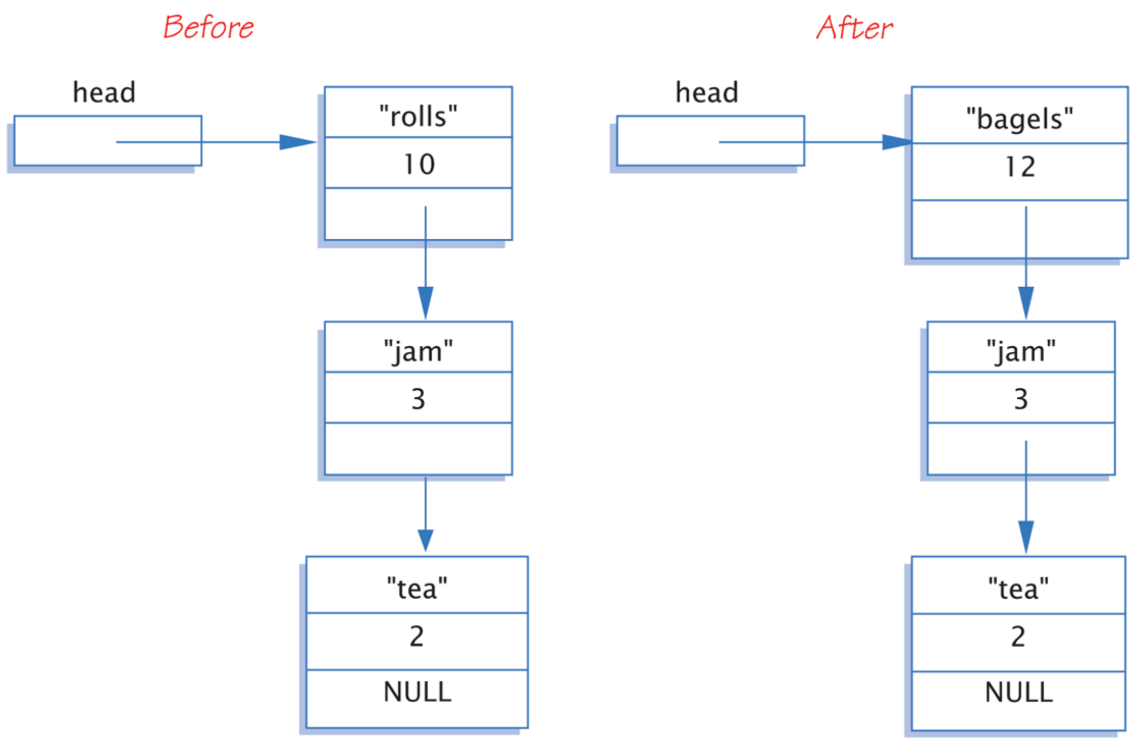
# End Markers

- Use NULL for node pointer
  - Considered "sentinel" for nodes
  - Indicates no further "links" after this node
- Provides end marker similar to how we use partially-filled arrays

# **Display 17.2** Accessing Node Data

**Display 17.2 Accessing Node Data**

```
head->count = 12;
head->item = "bagels";
```



# Linked List

- Lists as illustrated called linked lists
- First node called *head*
  – Pointed to by pointer named *head*
- Last node special also
  – Its member pointer variable is NULL
  – Easy test for "end" of linked list

# Linked List Class Definition

- class IntNode

```
{
public:
        IntNode() { }
        IntNode(int theData, IntNode* theLink)
                : data(theData), link(theLink) { }
        IntNode* getLink()          const          {return link;}
        int getData()               const          {return data;}
        void setData(int theData)             {data = theData;}
        void setLink(IntNode* pointer)        {link=pointer;}
private:
        int data;
        IntNode *link;
};
typedef IntNode* IntNodePtr;
```

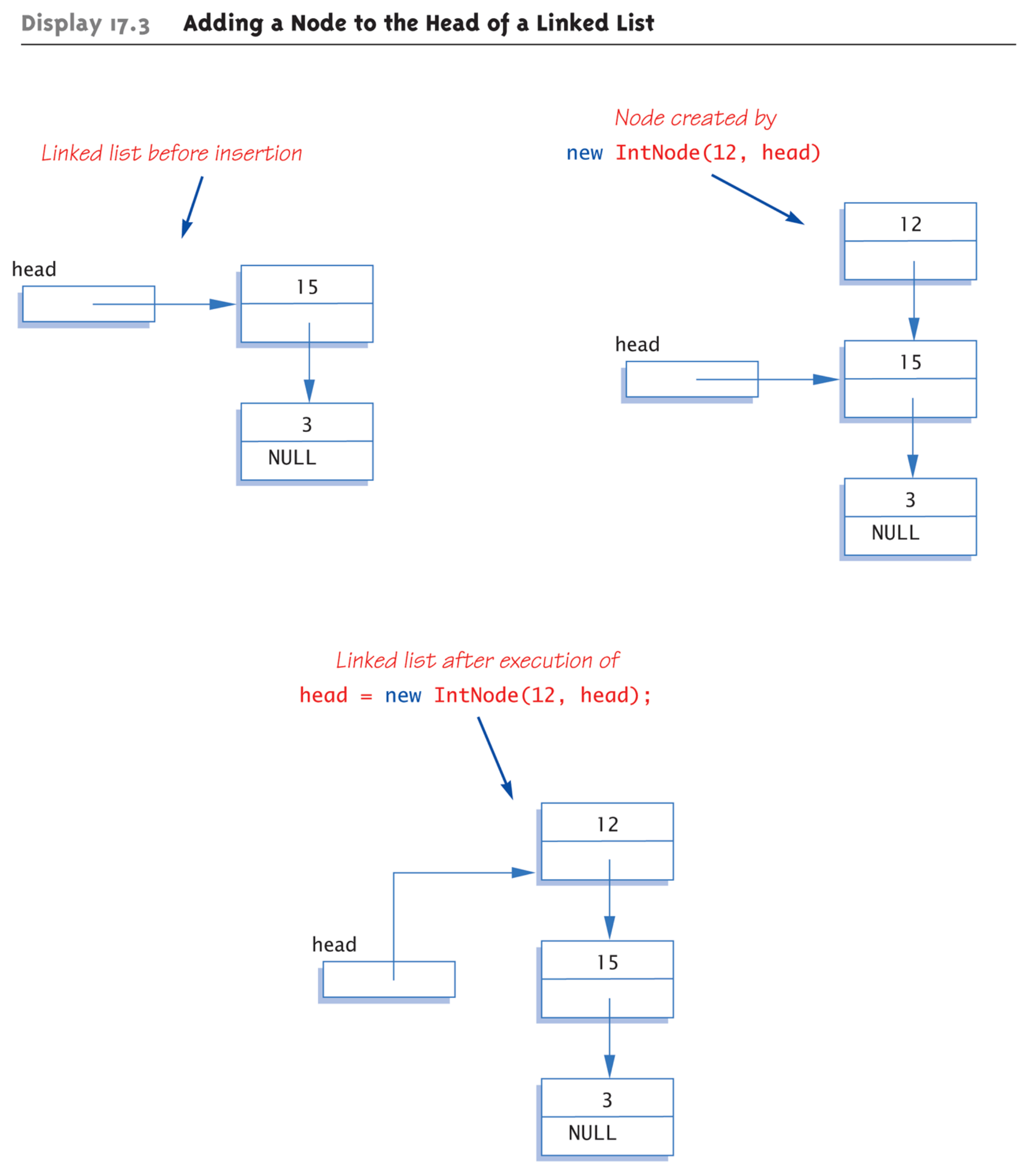
# Linked List Class

- Notice all member function definitions are inline
  - Small and simple enough
- Notice two-parameter constructor
  - Allows creation of nodes with specific data value and specified link member
  - Example:
    `IntNodePtr p2 = new IntNode(42, p1);`

# Create 1st Node

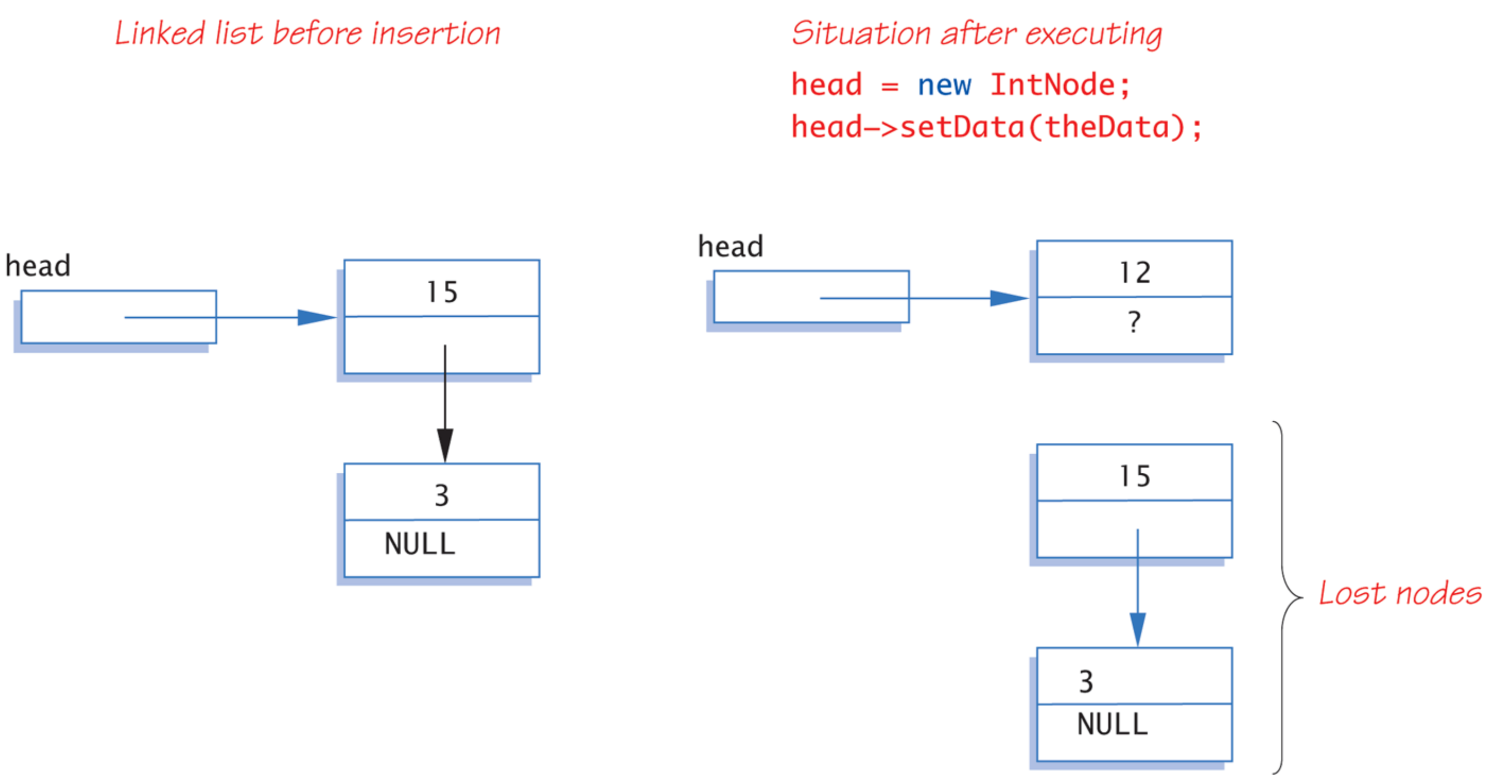
- IntNodePtr head;
  - Declares pointer variable *head*
- head = new IntNode;
  - Dynamically allocates new node
  - Our 1st node in list, so assigned to head
- head->setData(3);
  head->setLink(NULL);
  - Sets head node data
  - Link set to NULL since it's the only node!

# **Display 17.3** Adding a Node to the Head of a Linked List



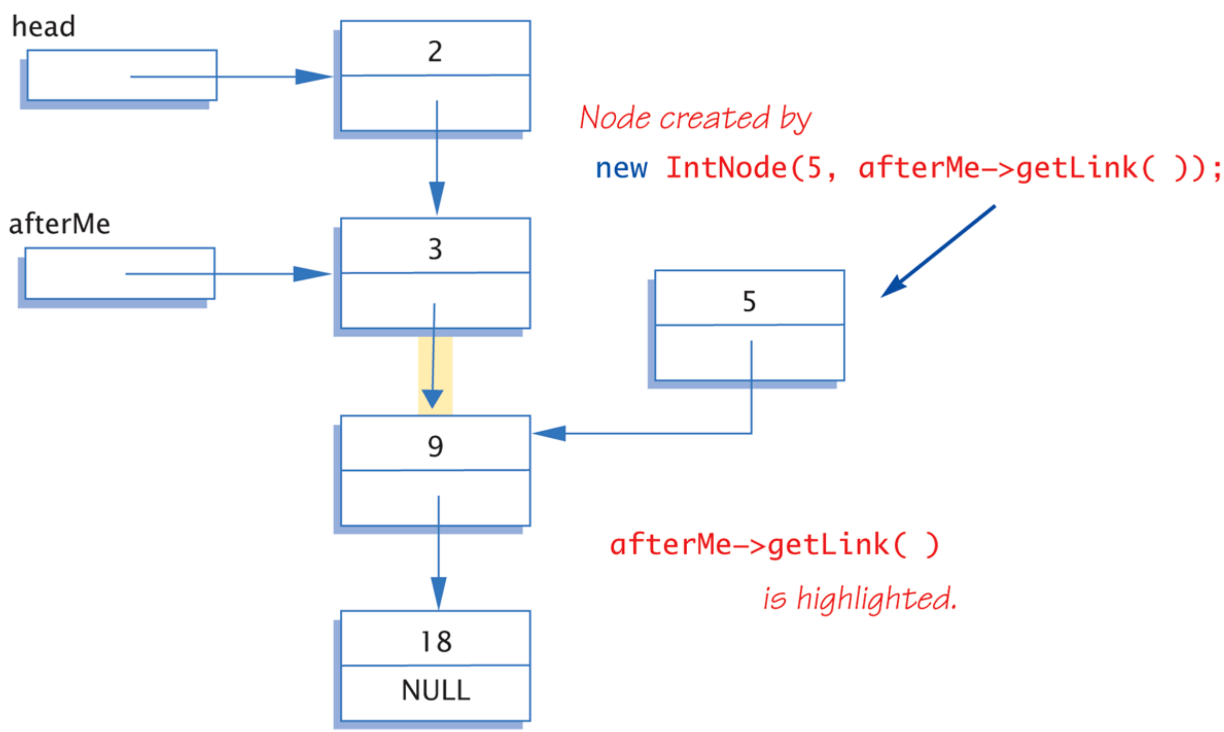
# Lost Nodes Pitfall: **Display 17.5** Lost Nodes

**Display 17.5 Lost Nodes**

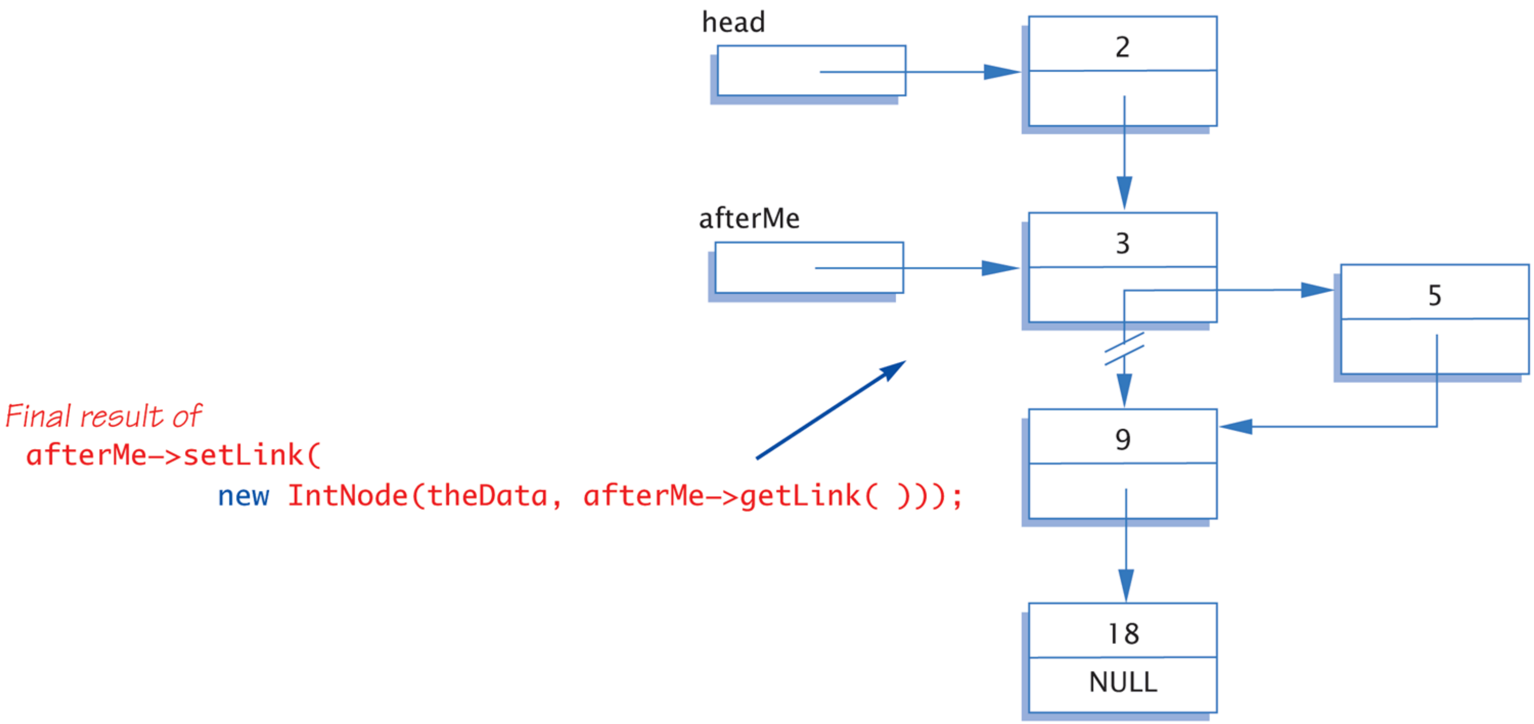


# **Display 17.6** Inserting in the Middle of a Linked List (1 of 2)

**Display 17.6 Inserting in the Middle of a Linked List**

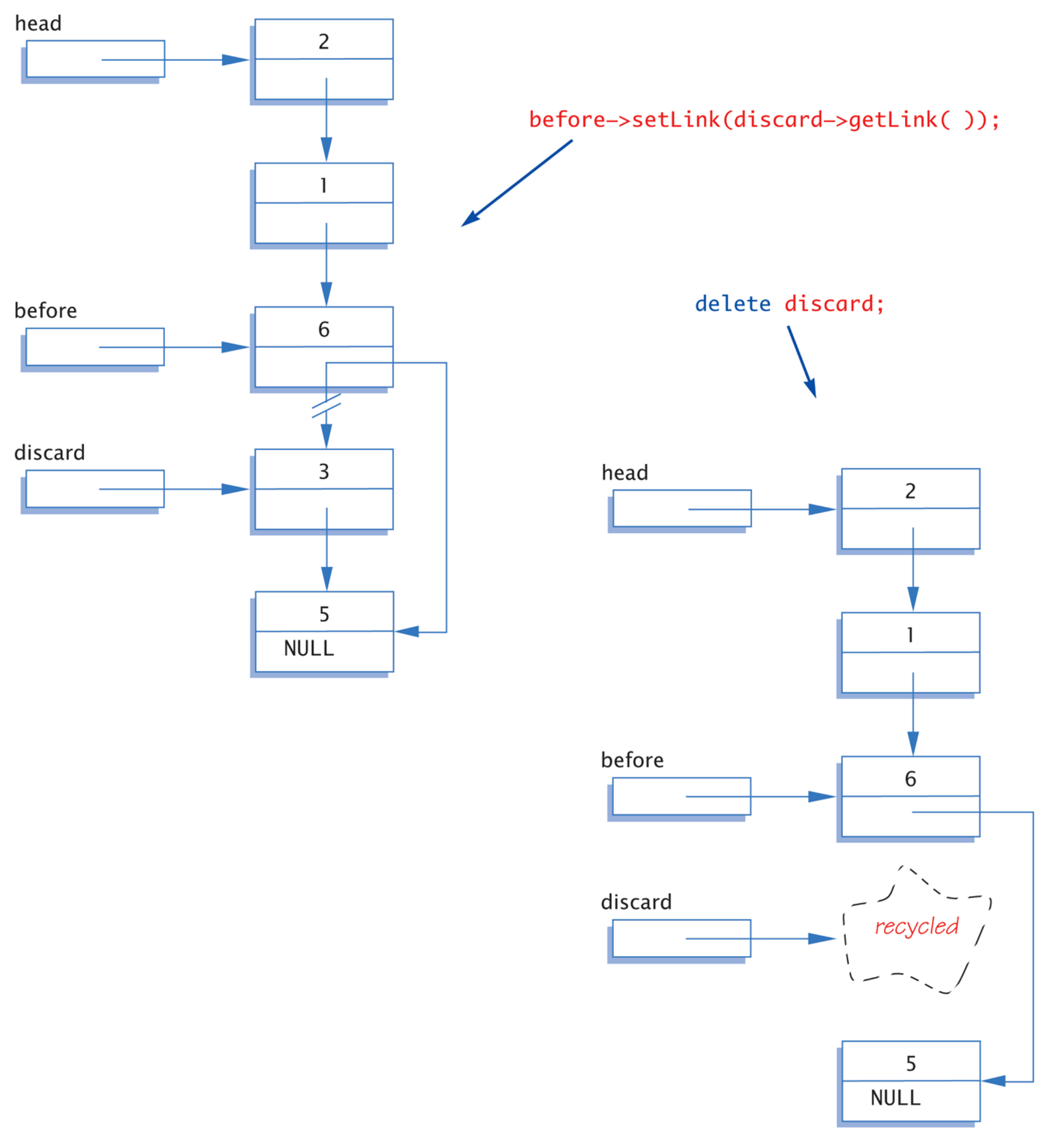


# **Display 17.6** Inserting in the Middle of a Linked List (2 of 2)



# Display 17.7 Removing a Node

Display 17.7 **Removing a Node**



# Searching a Linked List

- Function with two arguments:

  IntNodePtr search(IntNodePtr head, int target);

  **//Precondition: pointer head points to head of**
  **//linked list.  Pointer in last node is NULL.**
  **//If list is empty, head is NULL**
  **//Returns pointer to 1st node containing target**
  **//If not found, returns NULL**

- Simple "traversal" of list
  - Similar to array traversal

# Pseudocode for search Function

- while (here doesn't point to target node or
  last node)

```
{
    Make here point to next node in list
}
if (here node points to target)
    return here;
else
    return NULL;
```

# Algorithm for search Function

- while (here->getData() != target &&
                    here->getLink() != NULL)
          here = here->getLink();

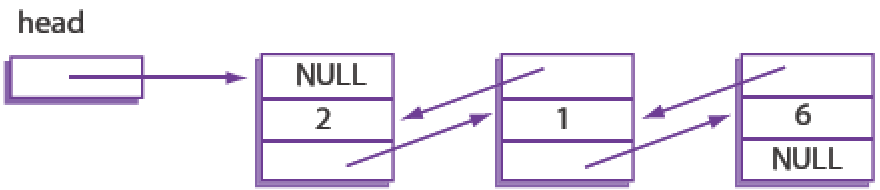
  if (here->getData() == target)
          return here;
  else
          return NULL;

- Must make "special" case for empty list
  - Not done here

# Doubly Linked Lists

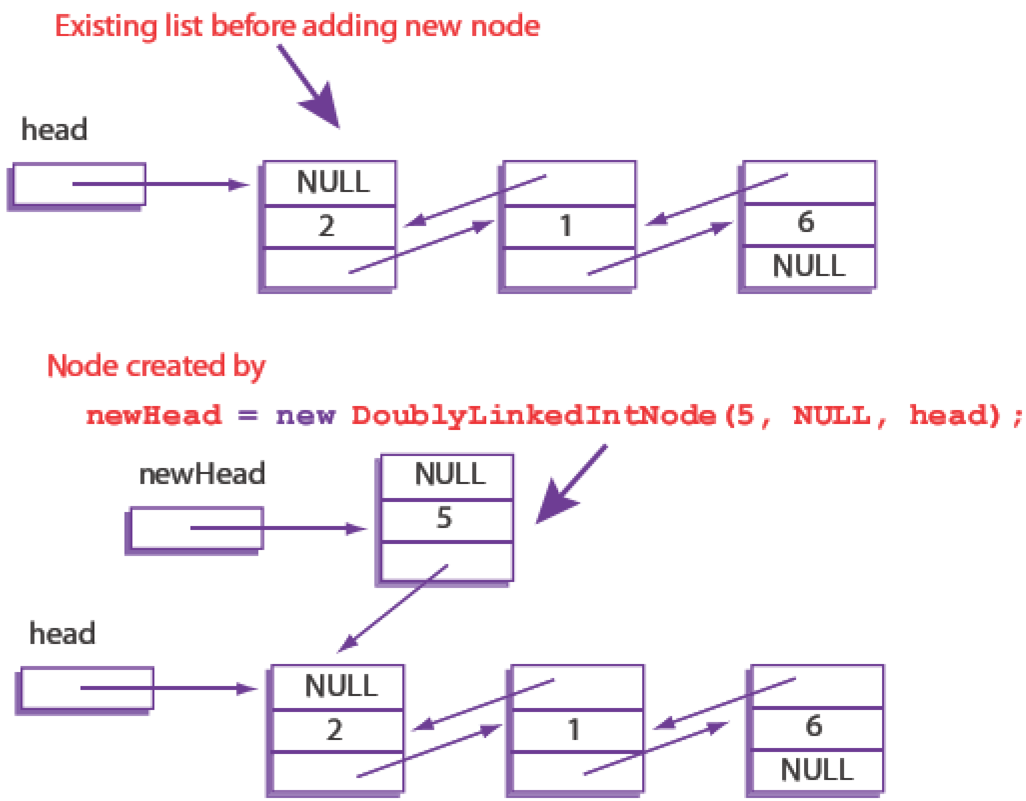
- What we just described is a singly linked list
  - Can only follow links in one direction
- Doubly Linked List
  - Links to the next node and another link to the previous node
  - Can follow links in either direction
  - NULL signifies the beginning and end of the list
  - Can make some operations easier, e.g. deletion since we don’t need to search the list to find the node before the one we want to remove

# Doubly Linked Lists

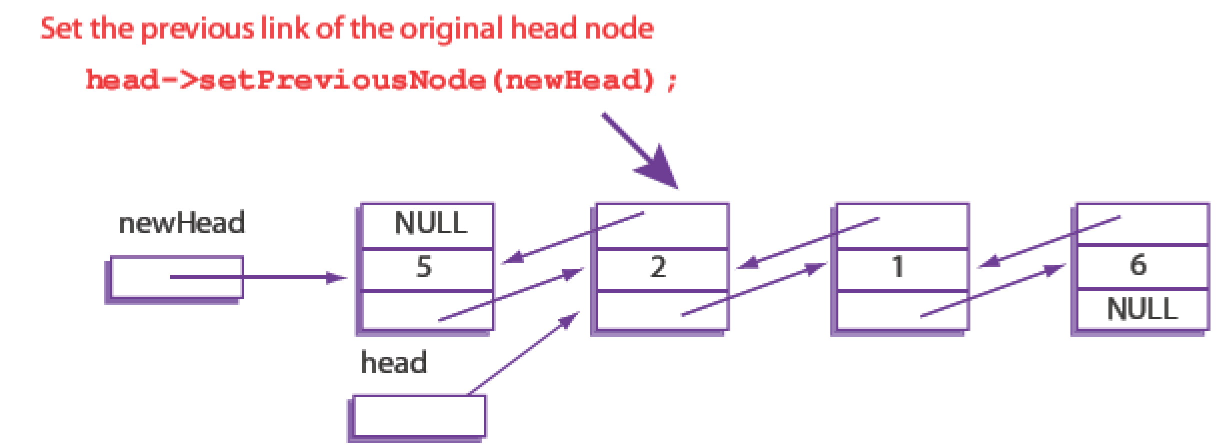


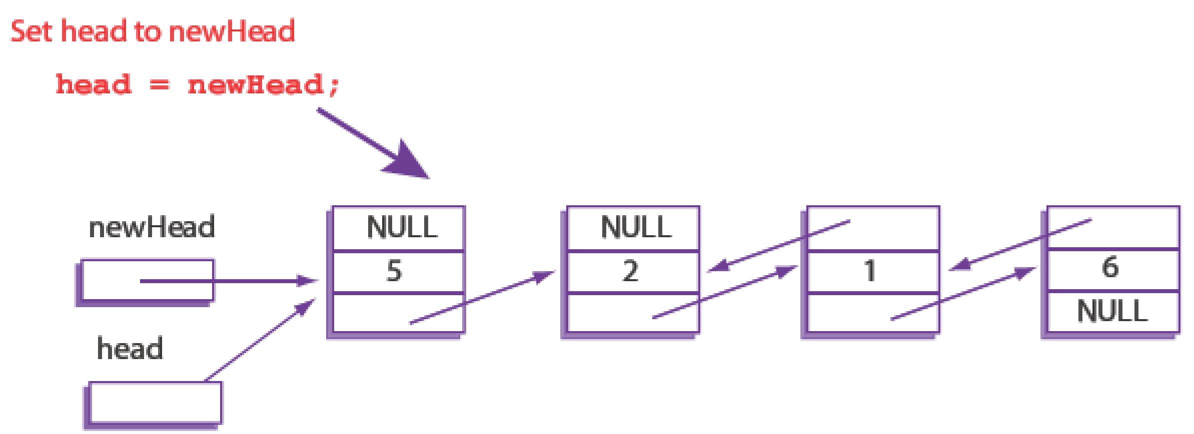

```
class DoublyLinkedIntNode
{
public:
    DoublyLinkedIntNode ( ){}
    DoublyLinkedIntNode (int theData, DoublyLinkedIntNode* previous,
                         DoublyLinkedIntNode* next)
            : data(theData), nextLink(next), previousLink(previous) {}
    DoublyLinkedIntNode* getNextLink( ) const { return nextLink; }
    DoublyLinkedIntNode* getPreviousLink( ) const { return previousLink; }
    int getData( ) const { return data; }
    void setData(int theData) { data = theData; }
    void setNextLink(DoublyLinkedIntNode* pointer) { nextLink = pointer; }
    void setPreviousLink(DoublyLinkedIntNode* pointer)
         { previousLink = pointer; }
private:
    int data;
    DoublyLinkedIntNode *nextLink;
    DoublyLinkedIntNode *previousLink;
};
typedef DoublyLinkedIntNode* DoublyLinkedIntNodePtr;
```

# Adding a Node to the Front of a Doubly Linked List (1 of 2)



# Adding a Node to the Front of a Doubly Linked List (2 of 2)



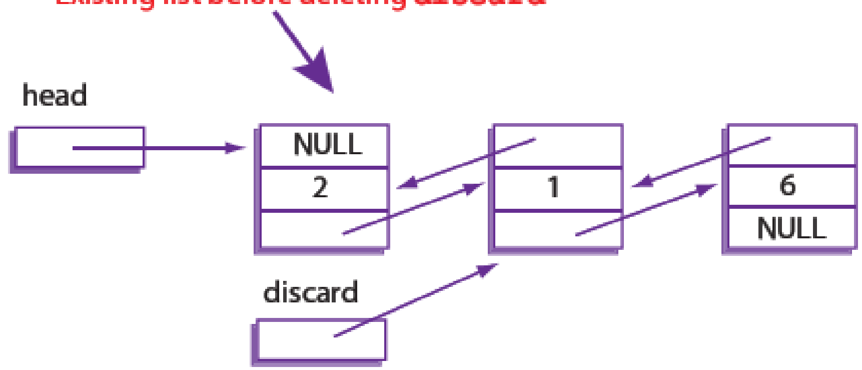



# Deleting a Node from a Doubly Linked List

- Removing a node requires updating references on both sides of the node we wish to delete
- Thanks to the backward link we do not need a separate variable to keep track of the previous node in the list like we did for the singly linked list
  - Can access via node->previous

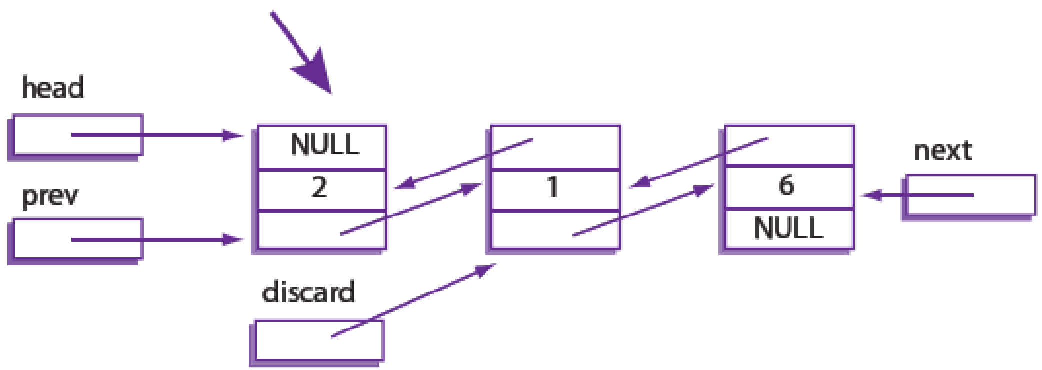
# Deleting a Node from a Doubly Linked List (1 of 2)

Existing list before deleting discard

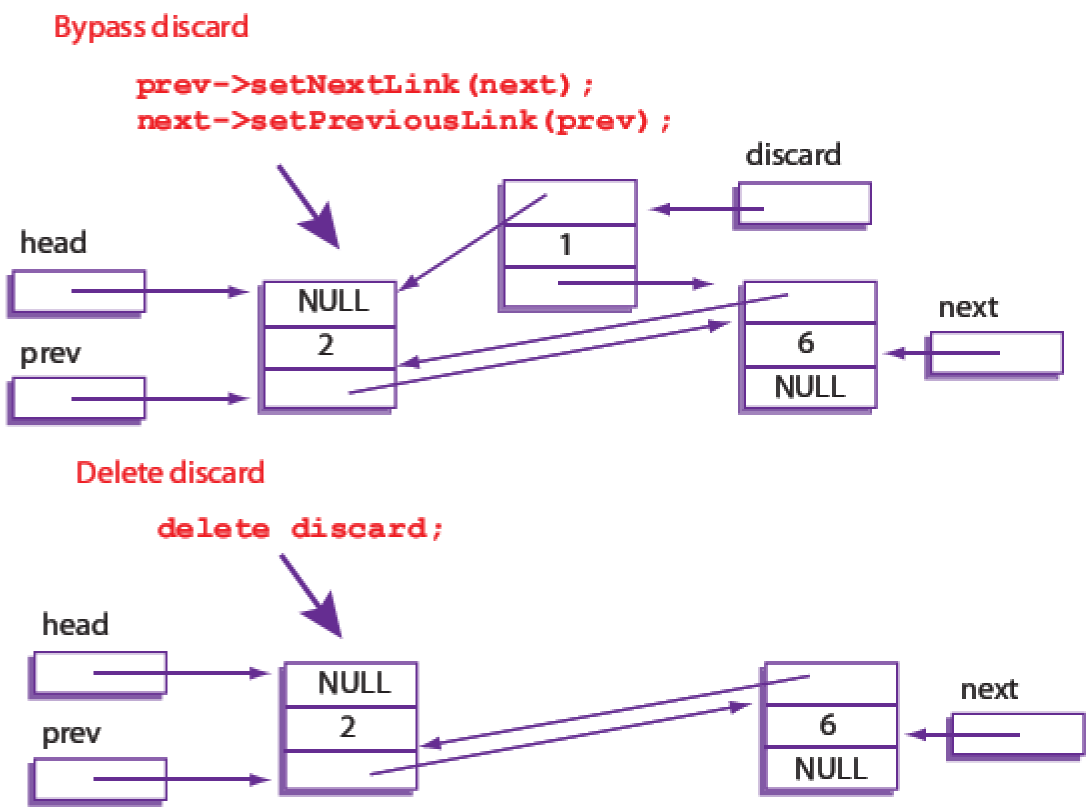



Set pointers to the previous and next nodes

```
DoublyLinkedIntNodePtr prev = discard->getPreviousLink( );
DoublyLinkedIntNodePtr next = discard->getNextLink( );
```

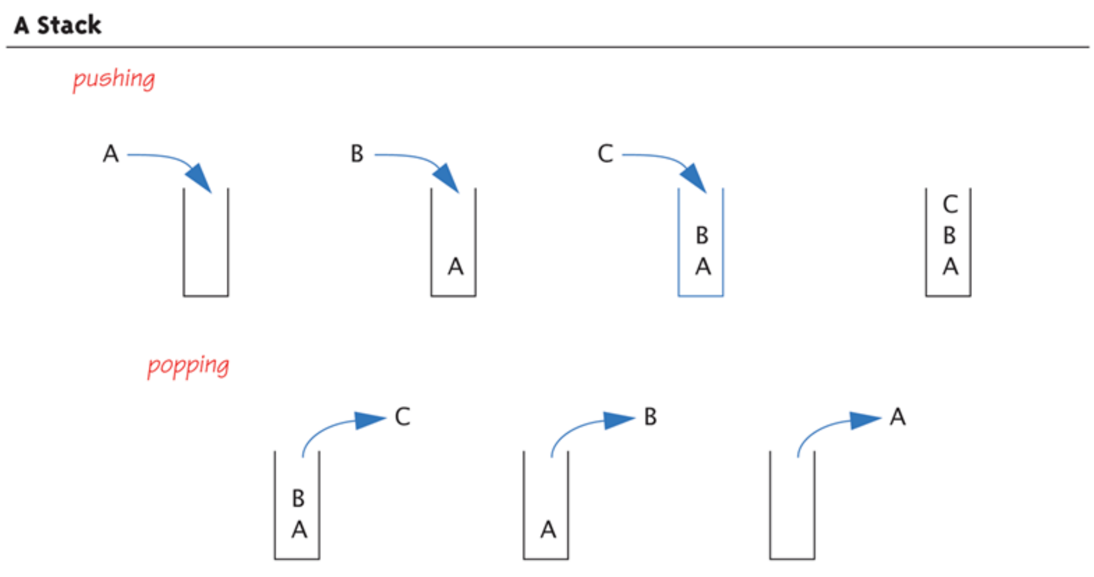


# Deleting a Node from a Doubly Linked List (2 of 2)



# Stacks

- Stack data structure:
  - Retrieves data in reverse order of how stored
  - LIFO – last-in/first-out
- Our use:
  - Use linked lists to implement stacks



# **Display 17.17** Interface File for a Stack Template Class (1 of 2)

**Interface File for a Stack Template Class**

```
//This is the header file stack.h. This is the interface for the class
//Stack, which is a template class for a stack of items of type T.
#ifndef STACK_H
#define STACK_H

namespace StackSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };
```

*You might prefer to replace the parameter type* T *with* const T&.

# **Display 17.17** Interface File for a Stack Template Class (2 of 2)

**Interface File for a Stack Template Class**

```
        template<class T>
    class Stack
    {
public:
        Stack( );
        //Initializes the object to an empty stack.

        Stack(const Stack<T>& aStack);

        Stack<T>& operator =(const Stack<T>& rightSide);

        virtual ~Stack( );

        void push(T stackFrame);
        //Postcondition: stackFrame has been added to the stack.

        T pop( );
        //Precondition: The stack is not empty.
        //Returns the top stack frame and removes that top
        //stack frame from the stack.

        bool isEmpty( ) const;
        //Returns true if the stack is empty. Returns false otherwise.
    private:
        Node<T> *top;
    };

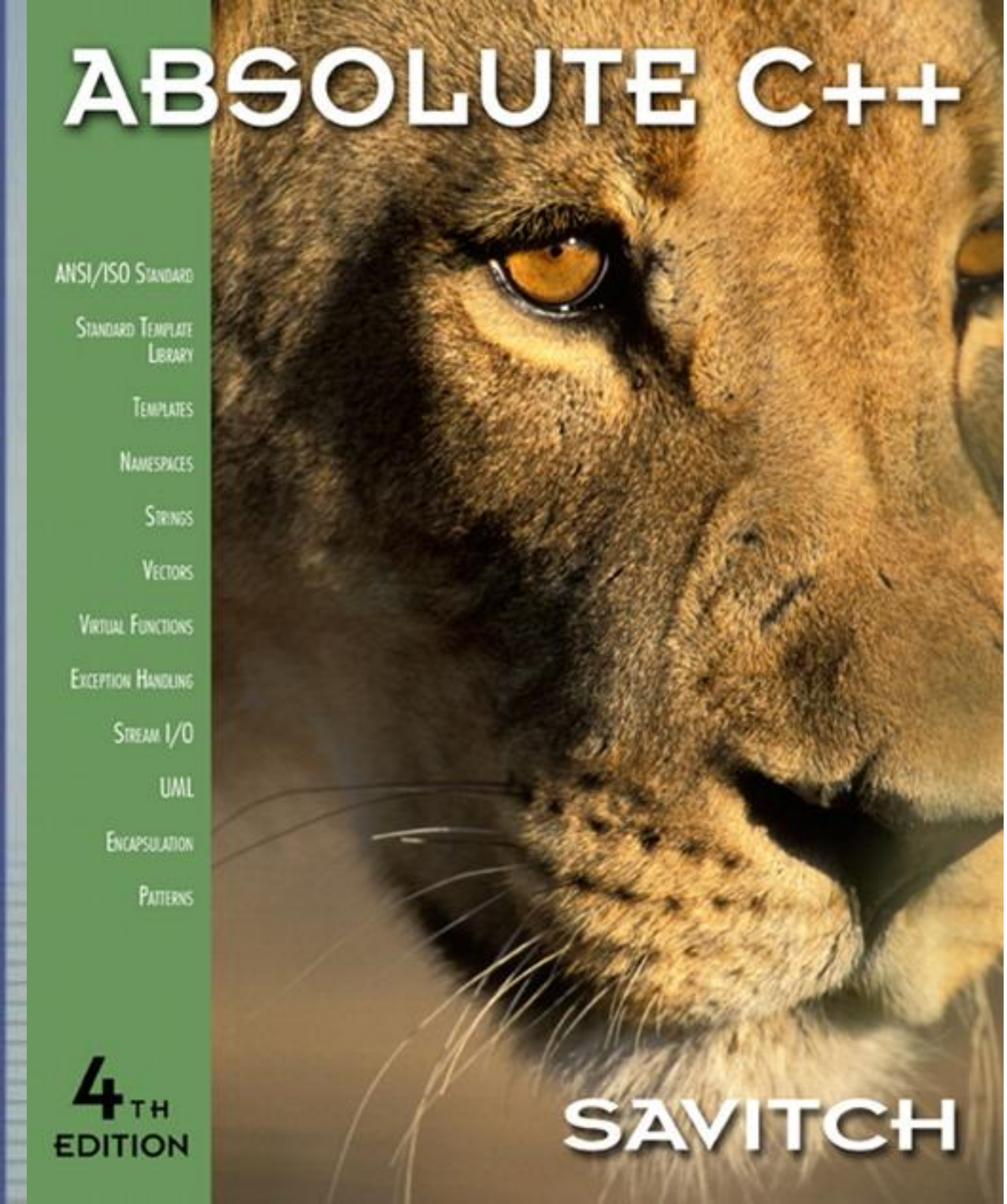
 }//StackSavitch
 #endif //STACK_H
```

*Copy constructor*

*The destructor destroys the stack and returns all the memory to the freestore.*

# Iterators

- Construct for cycling through data
  - Like a "traversal"
  - Allows "whatever" actions required on data
- For arrays, iteration is incrementing integer index
- For linked lists, iteration is pointer moving from one node to next
- We will see more with STL




# Chapter 7

## Constructors and Other Tools

## (like **STL vectors**)

# Learning Objectives

- Vectors
  - Introduction to vector class

# Vectors

- Vector Introduction
  - Recall: arrays are fixed size
  - Vectors: "arrays that grow and shrink"
    - During program execution
  - Formed from Standard Template Library (STL)
    - Using template class

# Vector Basics

- Similar to array:
  - Has base type
  - Stores collection of base type values
- Declared differently:
  - Syntax: vector<Base_Type>
    - Indicates template class
    - Any type can be "plugged in" to Base_Type
    - Produces "new" class for vectors with that type
  - Example declaration:
    vector<int> v;

# Vector Use

- vector<int> v;
  - "v is vector of type int"
  - Calls class default constructor
    - Empty vector object created
- Indexed like arrays for access
- But to add elements:
  - Must call member function push_back
- Member function size()
  - Returns current number of elements

# Vector Example:
# **Display 7.7** Using a Vector (1 of 2)

**Display 7.7** **Using a Vector**

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

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }
```

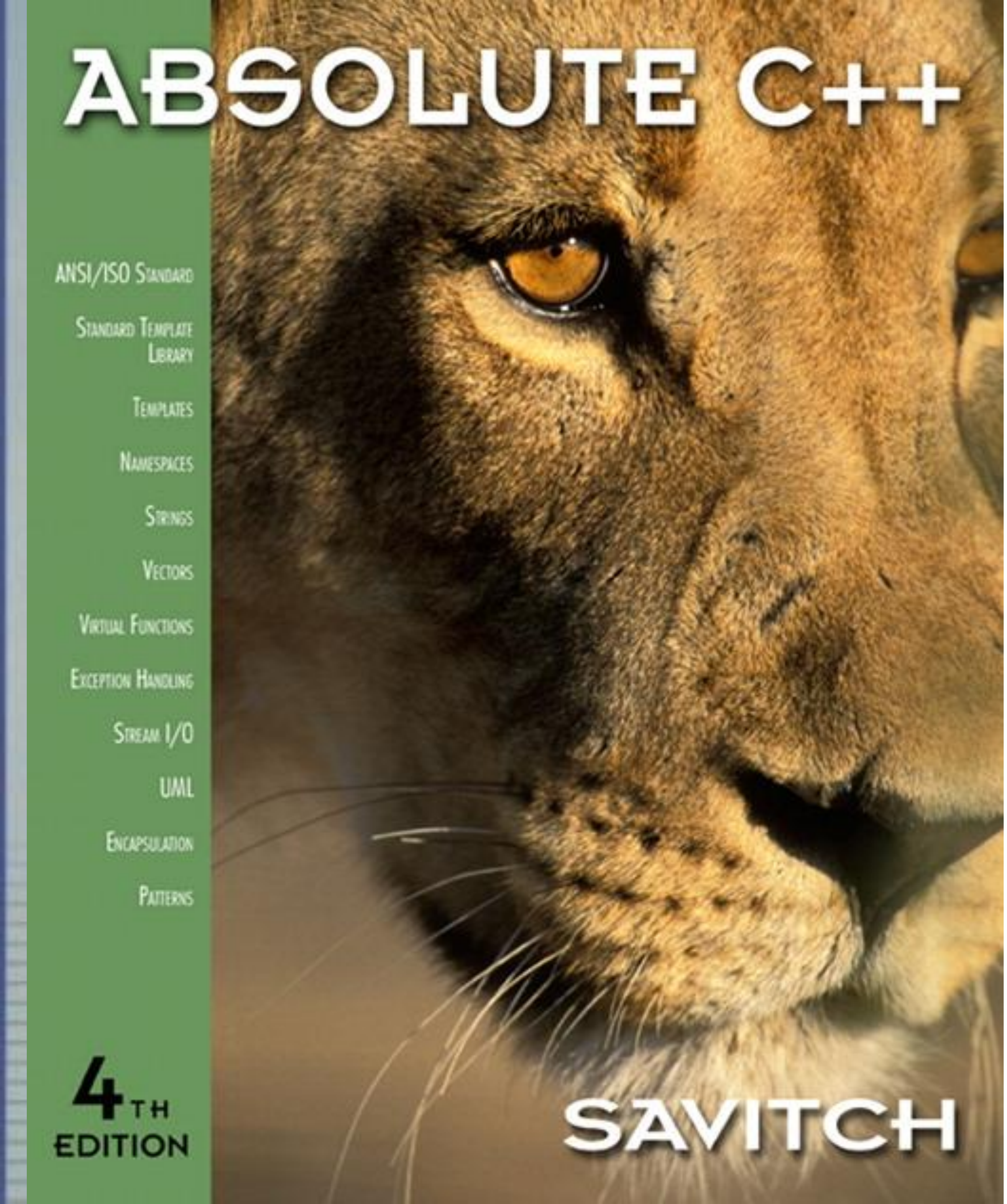
# Vector Example: **Display 7.7** Using a Vector (2 of 2)

```
    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

**SAMPLE DIALOGUE**

Enter a list of positive numbers.
Place a negative number at the end.
**2 4 6 8 –1**
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8




# Chapter 19

## Standard Template Library

# Learning Objectives

- Iterators
  - Reverse iterators
- Containers
  - Sequential containers
  - Container adapter stack
  - Associative Containers set and map

# Introduction

- Recall stack data structure
  - We created our own
  - Large collection of standard data structures exists
  - Make sense to have standard portable implementations of them!
- Standard Template Library (STL)
  - Includes libraries for all such data structures
    - Like container classes: stacks

# Iterators

- Recall: generalization of a pointer
  - Typically even implemented with pointer!
- "Abstraction" of iterators
  - Designed to hide details of implementation
  - Provide uniform interface across different container classes
- Each container class has "own" iterator type
  - Similar to how each data type has own pointer type

# Manipulating Iterators

- Recall using overloaded operators:
  - ++, --, ==, !=
  - *
    - So if p is iterator variable, *p gives access to data pointed to by p
- Vector template class
  - Has all above overloads
  - Also has members begin() and end()

```
c.begin();          //Returns iterator for 1st item in c
c.end();            //Returns "test" value for end
```

# Cycling with Iterators

- Recall cycling ability:

```
for (p=c.begin();p!=c.end();p++)
      process *p   //*p is current data item
```

- Big picture so far…
- Keep in mind:
  – Each container type in STL has own iterator types
    - Even though they're all used similarly

# Display 19.1
Iterators Used with a Vector (1 of 2)

```
//Program to demonstrate STL iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main( )
{
    vector<int> container;

    for (int i = 1; i <= 4; i++)
        container.push_back(i);

    cout << "Here is what is in the container:\n";
    vector<int>::iterator p;
    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Setting entries to 0:\n";
    for (p = container.begin( ); p != container.end( ); p++)
         *p = 0;
```

# Display 19.1

## Iterators Used with a Vector (2 of 2)

```
        cout << "Container now contains:\n";
        for (p = container.begin( ); p !=
                container.end( ); p++)
            cout << *p << " ";
        cout << endl;

        return 0;
    }
```

**SAMPLE DIALOGUE**

Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0

# Vector Iterator Types

- Iterators for vectors of ints are of type:
  std::vector<int>::iterator
- Iterators for lists of ints are of type:
  std::list<int>::iterator

- Vector is in std namespace, so need:
  using std::vector<int>::iterator;

# Kinds of Iterators

- Different containers → different iterators
- Vector iterators
  - Most "general" form
  - All operations work with vector iterators
  - Vector container great for iterator examples

# Random Access: **Display 19.2** Bidirectional and Random-Access Iterator Use

```
int main( )
{
    vector<char> container;

    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    container.push_back('D');

    for (int i = 0; i < 4; i++)
        cout << "container[" << i << "] == "
             << container[i] << endl;

    vector<char>::iterator p = container.begin( );
    cout << "The third entry is " << container[2] << endl;
    cout << "The third entry is " << p[2] << endl;
    cout << "The third entry is " << *(p + 2) << endl;

    cout << "Back to container[0].\n";
    p = container.begin( );
    cout << "which has value " << *p << endl;

    cout << "Two steps forward and one step back:\n";
    p++;
    cout << *p << endl;
```

*Three different notations for the same thing*

*This notation is specialized to vectors and arrays.*

*These two work for any random-access iterator.*

# Iterator Classifications

- Forward iterators:
  - ++ works on iterator
- Bidirectional iterators:
  - Both ++ and – work on iterator
- Random-access iterators:
  - ++, --, and random access [] all work with iterator
- These are "kinds" of iterators, not types!

# Constant and Mutable Iterators

- Dereferencing operator's behavior dictates
- Constant iterator:
  - * produces read-only version of element
  - Can use *p to assign to variable or output, but cannot change element in container
    - E.g., *p = <anything>; is illegal
- Mutable iterator:
  - *p can be assigned value
  - Changes corresponding element in container
  - i.e.: *p returns an lvalue

# Reverse Iterators

- To cycle elements in reverse order
  - Requires container with bidirectional iterators
- Might consider:

```
iterator p;
for (p=container.end();p!=container.begin(); p--)
      cout << *p << " " ;
```

  - But recall: end() is just "sentinel", begin() not!
  - Might work on some systems, but not most

# Reverse Iterators Correct

- To correctly cycle elements in reverse order:

```
reverse_iterator p;
for (rp=container.rbegin();rp!=container.rend(); rp++)
        cout << *rp << " " ;
```

- rbegin()
  - Returns iterator at last element
- rend()
  - Returns sentinel "end" marker

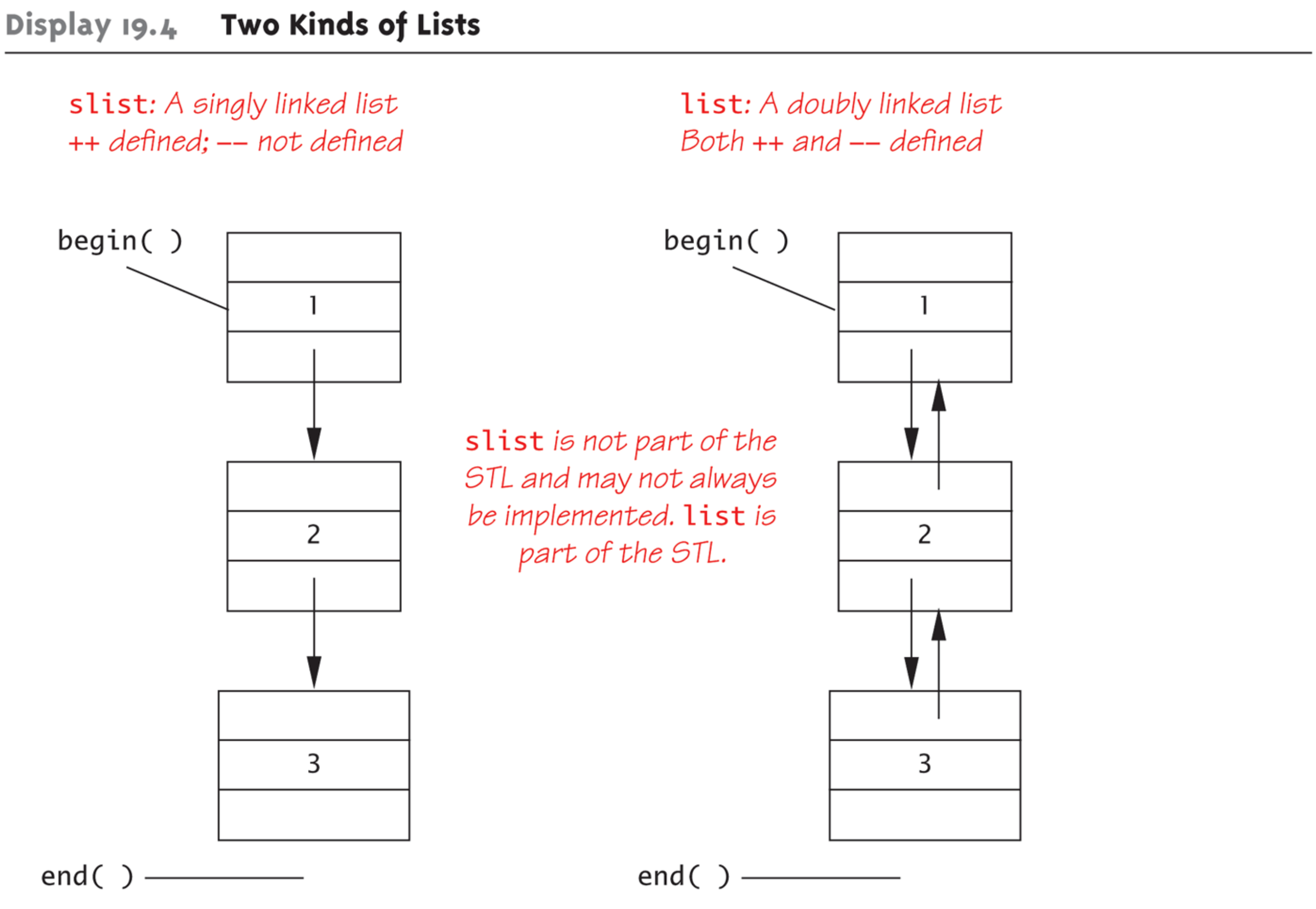
# Containers

- Container classes in STL
  - Different kinds of data structures
  - Like lists, queues, stacks
- Each is template class with parameter for particular data type to be stored
  - e.g., Lists of ints, doubles or myClass types
- Each has own iterators
  - One might have bidirectional, another might just have forward iterators
- But all operators and members have same meaning

# Sequential Containers

- Arranges list data
  - 1st element, next element, … to last element
- Linked list is sequential container
  - Earlier linked lists were "singly linked lists"
    - One link per node
- STL has no "singly linked list"
  - Only "doubly linked list": template class *list*

# **Display 19.4** Two Kinds of Lists



# Display 19.5
## Using the list Template Class(1 of 2)

```
//Program to demonstrate the STL template class list.
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main( )
{
    list<int> listObject;

    for (int i = 1; i <= 3; i++)
        listObject.push_back(i);

    cout << "List contains:\n";
    list<int>::iterator iter;
    for (iter = listObject.begin( ); iter != listObject.end( );
          iter++)
        cout << *iter << " ";
    cout << endl;
```

# Display 19.5
Using the list Template Class(2 of 2)

```
        cout << "Setting all entries to 0:\n";
        for (iter = listObject.begin( ); iter != listObject.end( );
                          iter++)
            *iter = 0;

        cout << "List now contains:\n";
        for (iter = listObject.begin( ); iter != listObject.end( );
                          iter++)
            cout << *iter << " ";
        cout << endl;

        return 0;
    }
```

**SAMPLE DIALOGUE**

```
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
```

# Container Adapter *stack*

- Container adapters are template classes
  - Implemented "on top of" other classes
- Example:
  *stack* template class by default implemented on top of *deque* template class
  - Buried in stack's implementation is deque where all data resides
  - top() returns reference to first item on stack without removing it
  - pop() removes it without returning a reference

# Associative Containers

- Associative container: simple database
- Store data
  - Each data item has key
- Example:
  data: employee's record as struct
  key: employee's SSN
  - Items retrieved based on key

# set Template Class

- Simplest container possible
- Stores elements without repetition
- 1st insertion places element in set
- Each element is own key
- Capabilities:
  – Add elements
  – Delete elements
  – Ask if element is in set

# Program Using the set Template Class (1 of 2)

```
//Program to demonstrate use of the set template class.
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;

int main( )
{
    set<char> s;

    s.insert(’A’);
    s.insert(’D’);
    s.insert(’D’);
    s.insert(’C’);
    s.insert(’C’);
    s.insert(’B’);

    cout << "The set contains:\n";
    set<char>::const_iterator p;
    for (p = s.begin( ); p != s.end( ); p++)
    cout << *p << " ";
    cout << endl;
```

# Program Using the set Template Class (2 of 2)

```
     cout << "Set contains 'C': ";
     if (s.find('C')==s.end( ))
        cout << " no " << endl;
     else
        cout << " yes " << endl;

       cout << "Removing C.\n";
       s.erase(’C’);
       for (p = s.begin( ); p != s.end( ); p++)
       cout << *p << " ";
       cout << endl;

     cout << "Set contains 'C': ";
     if (s.find('C')==s.end( ))
         cout << " no " << endl;
     else
         cout << " yes " << endl;

       return 0;
   }
```

**SAMPLE DIALOGUE**

```
The set contains:
A B C D
Set contains 'C':  yes
Removing C.
A B D
Set contains 'C':  no
```

# Map Template Class

- A function given as set of ordered pairs
  - For each value first, at most one value second in map
- Example map declaration:
  map<string, int> numberMap;
- Can use [ ] notation to access the map
  - For both storage and retrieval
- Stores in sorted order, like set
  - Second value can have no ordering impact

# Program Using the map Template Class (1 of 3)

```
//Program to demonstrate use of the map template class.
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

int main( )
{
    map<string, string> planets;

    planets["Mercury"] = "Hot planet";
    planets["Venus"] = "Atmosphere of sulfuric acid";
    planets["Earth"] = "Home";
    planets["Mars"] = "The Red Planet";
    planets["Jupiter"] = "Largest planet in our solar system";
    planets["Saturn"] = "Has rings";
    planets["Uranus"] = "Tilts on its side";
    planets["Neptune"] = "1500 mile per hour winds";
    planets["Pluto"] = "Dwarf planet";
```

# Program Using the map Template Class (2 of 3)

```
        cout << "Entry for Mercury - " << planets["Mercury"]
                << endl << endl;

        if (planets.find("Mercury") != planets.end())
            cout << "Mercury is in the map." << endl;
        if (planets.find("Ceres") == planets.end())
            cout << "Ceres is not in the map." << endl << endl;

        cout << "Iterating through all planets: " << endl;
        map<string, string>::const_iterator iter;
        for (iter = planets.begin(); iter != planets.end(); iter++)
        {
            cout << iter->first << " - " << iter->second << endl;
        }
```

**The iterator will output the map in order sorted by the key. In this case the output will be listed alphabetically by planet.**

```
        return 0;
    }
```

# Program Using the map Template Class (3 of 3)

```
SAMPLE DIALOGUE

Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.

Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile per hour winds
Pluto - Dwarf planet
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid
```