Programación orientada a objetos en C++

Pedro Corcuera

Dpto. Matemática Aplicada y Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Table of Contents

1. Introduction to Classes
2. Inheritance, Polymorphism, and Virtual Functions
3. Exceptions and Templates
4. The Standard Template Library
5. Data Structures

1. Introduction to Classes

Procedural and Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program, with function as the basic unit. You need to first figure out all the functions and then think about how to represent data.

- Object-Oriented programming is based on the data and the functions that operate on it. Objects are instances of ADTs that represent the data and its functions

Limitations of Procedural Programming

- If the data structures change, many functions must also be changed
- Programs that are based on complex function hierarchies are:
  - difficult to understand and maintain
  - difficult to modify and extend
  - easy to break

Benefits of OOP

- The object-oriented languages focus on components that the user perceives, with objects as the basic unit. You figure out all the objects by putting all the data and operations that describe the user's interaction with the data.
- Object-Oriented technology has many benefits:
  - *Ease in software design*: You are dealing with high-level concepts and abstractions. Ease in design leads to more productive software development.
  - *Ease in software maintenance*: object-oriented software are easier to understand, therefore easier to test, debug, and maintain.
  - *Reusable software*: you don't need to keep re-inventing the wheels and re-write the same functions for different situations. The fastest and safest way of developing a new application is to reuse existing codes - fully tested and proven codes.

Object-Oriented Programming Terminology

- **class**: a class is a definition of objects of the same kind. A class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviors common to all objects of the same kind.
- **instance**: An instance is a realization of a particular item of a class. An instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition.
- **object**: an instance of a `class`, in the same way that a variable can be an instance of a `struct`
  - **attributes**: members of a class
  - **methods** or behaviors: member functions of a class

Object-Oriented Programming Languages

- OOP languages permit higher level of abstraction for solving real-life problems. The traditional procedural language (C and Pascal) forces you to think in terms of the structure of the computer (memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve.
- The OOP languages (Java, C++, C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.

Object-Oriented Programming

Name
Attributes
Behaviors

Name
Attributes
Behaviors

Name
Attributes
Behaviors

messages

Name
Attributes
Behaviors

Name
Attributes
Behaviors

An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Classes and Objects

- **Class**: A class is a definition of objects of the same kind. A class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviors common to all objects of the same kind.
- **Instance**: An instance is a realization of a particular item of a class. An instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition.
- The term **object** usually refers to instance. But it is often used quite loosely, which may refer to a class or an instance.

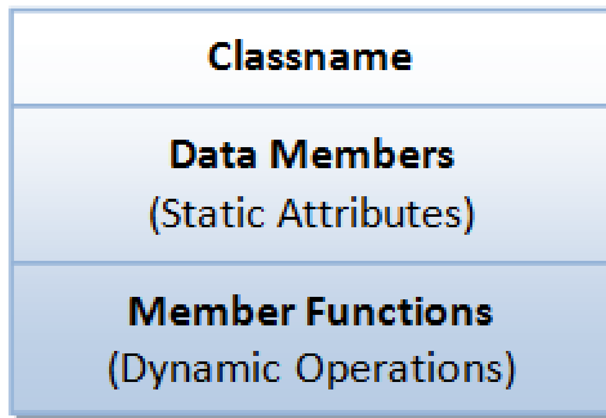
More on Objects

- data hiding: restricting access to certain members of an object
- public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

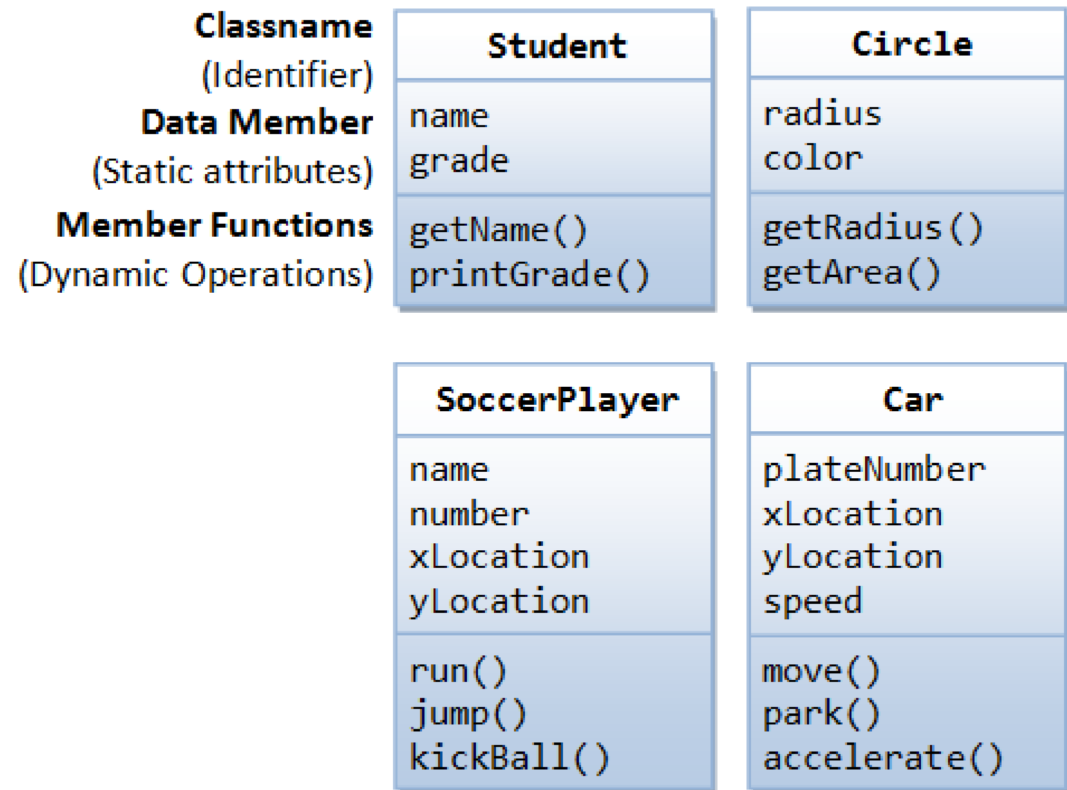
A Class is a 3-Compartment Box encapsulating Data and Functions

- A class can be visualized as a three-compartment box:
  - **Classname** (or identifier): identifies the class.
  - **Data Members** or **Variables** (or attributes, states, fields): contains the static attributes of the class.
  - **Member Functions** (or methods, behaviors, operations): contains the dynamic operations of the class.
- A class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.
- **Class Members**: The data members and member functions are collectively called class members.

A Class is a 3-Compartment Box encapsulating Data and Functions



A class is a 3-compartment box encapsulating data and functions



Examples of classes

The Unified Modeling Language

- *UML* stands for ***U**nified **M**odeling **L**anguage*.
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems. UML specification defines two major kinds of UML diagram: structure diagrams and behavior diagrams.
  - **Structure diagrams** show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.
  - **Behavior diagrams** show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

Classification of UML Diagrams

UML 2.5 Diagram

Structure Diagram

Behavior Diagram

Class Diagram

Object Diagram

Package Diagram

Model Diagram

Composite Structure Diagram

Internal Structure Diagram

Collaboration Use Diagram

Component Diagram

Manifestation Diagram

Deployment Diagram

Network Architecture Diagram

Profile Diagram

UseCase Diagram

Information Flow Diagram

Activity Diagram

State Machine Diagram

Behavioral State Machine Diagram

Protocol State Machine Diagram

Interaction Diagram

Sequence Diagram

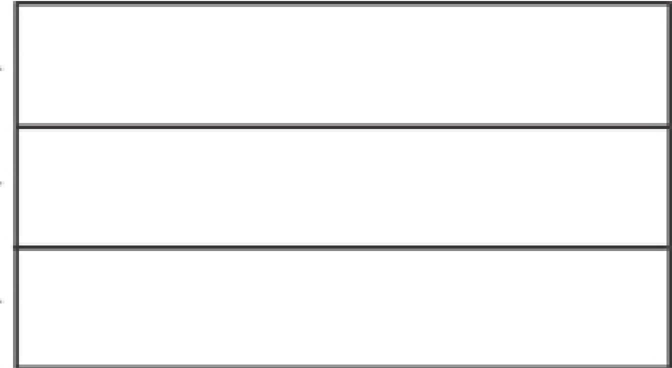
Communication Diagram

Timing Diagram

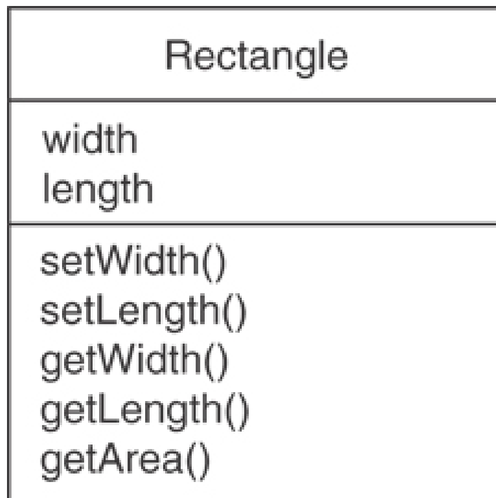
Interaction Overview Diagram

UML Class Diagram

- A UML diagram for a class has three main sections.



- Example



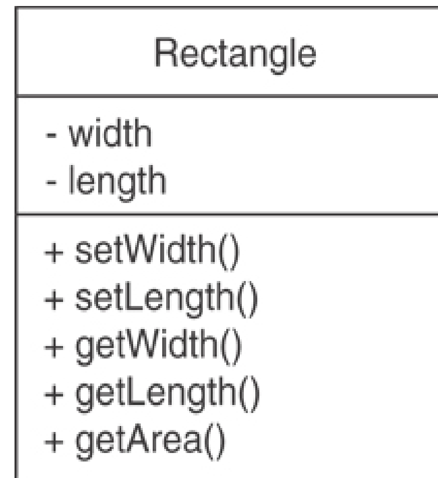

```
class Rectangle {
   private:
      double width; double length;
   public:
      bool setWidth(double);
      bool setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

UML Notation

- Access Specification
  - In UML you indicate a private member with a minus (-) and a public member with a plus(+)

These member variables are private.

These member functions are public.



- Data type
  - To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable.

```
- width : double
- length : double
```

UML Notation

- Parameter Type
  - To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable

```
+ setWidth(w : double)
```

- Function Return Type
  - To indicate the data type of a function's return value, place a colon followed by the name of the data type after the function's parameter list.

```
+ setWidth(w : double) : void
```

Classes

Rectangle Class

Rectangle
- width : double - length : double
+ setWidth(w : double) : bool + setLength(len : double) : bool + getWidth() : double + getLength() : double + getArea() : double

InventoryItem Class

InventoryItem
- description : char* - cost : double - units : int - createDescription(size : int, value : char*) : void
+ InventoryItem() : + InventoryItem(desc : char*) : + InventoryItem(desc : char*, c : double, u : int) : + ~InventoryItem() : + setDescription(d : char*) : void + setCost(c : double) : void + setUnits(u : int) : void + getDescription() : char* + getCost() : double + getUnits() : int

Constructors →

Destructor →

Class Definition

- Objects are created from a `class`
- Format:

```
class ClassName
{
  private:
      data member declaration;
  public:
      member functions declaration;
};
```

Class Example

```
class Rectangle
{
   private:
      double width;
      double length;
    public:
      Rectangle(double, double);
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

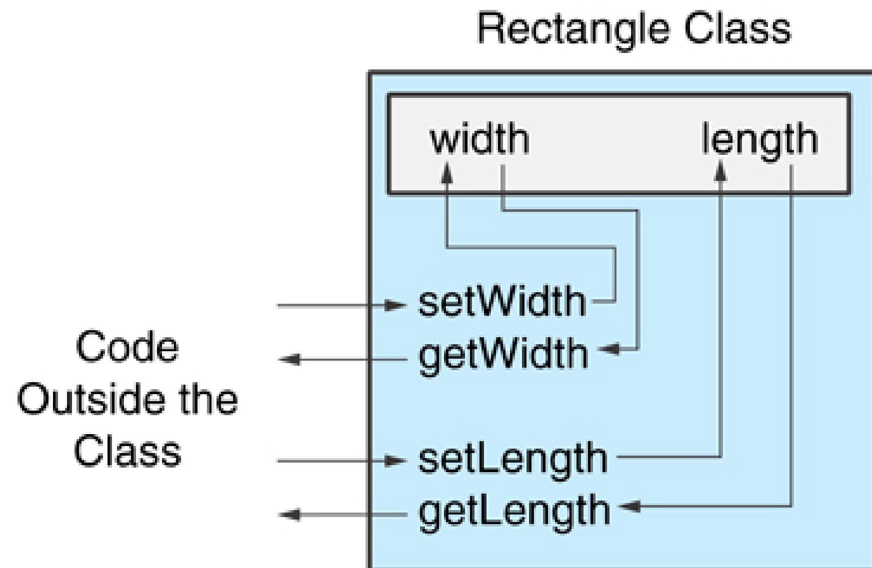
Private Members

Public Members

Why Have Private Members?

- Making data members `private` provides data protection
- Data can be accessed only through `public` functions
- Public functions define the class's public interface

Code outside the class must use the class's public member functions to interact with the object.



Classes in C++

- Access Specifiers:
  - Used to control access to members of the class
  - `public:` the member (data or function) is accessible and available to *all* in the system
  - `private:` the member (data or function) is accessible and available *within this class only*
  - Can be listed in any order in a class
  - Can appear multiple times in a class
  - If not specified, the default is `private`
- Using `const` with Member Functions
  - `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object

```
double getWidth() const;
double getLength() const;
double getArea() const;
```

Classes in C++

- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator (`::`)

```
int Rectangle::setWidth(double w)
{
    width = w;
}
```

- *Getters*: function that *read* the value of a private data member (ex. xxx named getXxx()). Getters do not change an object's data, so they should be marked `const`.
- *Setters*: function that *modify* the value of a private data member (ex. xxx setXxx)

Defining an Instance of a Class

- An object is an instance of a class
- To create an instance of a class, you have to:
  - Declare an instance (name) identifier of a particular class
  - Invoke a constructor to construct the instance (i.e., allocate storage for the instance and initialize the variables).
- Defined like structure variables:

```
Rectangle r;
```

Class Naming Convention

- A classname shall be a noun or a noun phrase made up of several words.
- All the words shall be initial-capitalized (camel-case).
- Use a singular noun for classname.
- Choose a meaningful and self-descriptive classname.
- Examples: Point, Rectangle, SoccerPlayer, HttpProxyServer, FileInputStream, PrintStream, SocketFactory.

Dot (.) Operator

- To reference a member of a object (data member or member function), you must:
  - First identify the instance you are interested in, and then
  - Use the dot operator (.) to reference the member, in the form of instanceName.memberName.
- Access members using dot operator:

```
r.setWidth(5.2);
cout << r.getWidth();
```

- Compiler error if attempt to access `private` member using dot operator

Keyword “this”

- You can use keyword "this" to refer to this instance inside a class definition
- One of the main usage of keyword this is to resolve ambiguity between the names of data member and function parameter. Example:

```
class Circle {
private:
   double radius; // Member variable "radius"
   ......
public:
   void setRadius(double radius) {
      this->radius = radius;
   }
   ......
}
```

Data Members (Variables)

- A data member (variable) has a name (or identifier) and a type; and holds a value of that particular type. A data member can also be an instance of a certain class.
- **Data Member Naming Convention**: A data member name shall be a noun or a noun phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). Example: fontSize, roomNumber, xMax, yMin and xTopLeft.
- Take note that variable name begins with an lowercase, while classname begins with an uppercase.

Member Functions

- A member function:
  - receives parameters from the caller,
  - performs the operations defined in the function body, and
  - returns a piece of result (or void) to the caller.
- **Member Function Naming Convention**: A function name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). Example: getRadius(), getParameterValues().
- Take note that data member name is a noun (denoting a static attribute), while function name is a verb (denoting an action). They have the same naming convention.

Program CircleAIO.cpp

```
/* The Circle class (All source codes in one file) (CircleAIO.cpp) */
#include <iostream>    // using IO functions
#include <string>      // using string
using namespace std;

class Circle {
private:
   double radius;      // Data member (Variable)
   string color;       // Data member (Variable)

public:
   // Constructor with default values for data members
   Circle(double r = 1.0, string c = "red") {
      radius = r;
      color = c;
   }

   double getRadius() {  // Member function (Getter)
      return radius;
   }

   string getColor() {   // Member function (Getter)
      return color;
   }

   double getArea() {    // Member function
      return radius*radius*3.1416;
   }
};   // need to end the class declaration with a semi-colon
```

Program CircleAIO.cpp

```
// Test driver function
int main() {
   // Construct a Circle instance
   Circle c1(1.2, "blue");
   cout << "Radius=" << c1.getRadius() << " Area=" << c1.getArea()
        << " Color=" << c1.getColor() << endl;

   // Construct another Circle instance
   Circle c2(3.4); // default color
   cout << "Radius=" << c2.getRadius() << " Area=" << c2.getArea()
        << " Color=" << c2.getColor() << endl;

   // Construct a Circle instance using default no-arg constructor
   Circle c3;      // default radius and color
   cout << "Radius=" << c3.getRadius() << " Area=" << c3.getArea()
        << " Color=" << c3.getColor() << endl;
   return 0;
}
```

Program Rectangle.cpp

```
// This program demonstrates a simple class.
#include <iostream>
using namespace std;

// Rectangle class declaration.
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
// setWidth assigns a value to the width member.   *
void Rectangle::setWidth(double w) {
   width = w;
}

// setLength assigns a value to the length member. *
void Rectangle::setLength(double len) {
   length = len;
}
// getWidth returns the value in the width member. *
double Rectangle::getWidth() const {
   return width;
}
```

Program Rectangle.cpp

```
// getLength returns the value in the length member. *
double Rectangle::getLength() const {
   return length;
}
// getArea returns the product of width times length. *
double Rectangle::getArea() const {
   return width * length;
}
// Function main                                       *
int main()
{
   Rectangle box;     // Define an instance of the Rectangle class
   double rectWidth;  // Local variable for width
   double rectLength; // Local variable for length
   // Get the rectangle's width and length from the user.
   cout << "This program will calculate the area of a\n";
   cout << "rectangle. What is the width? ";
   cin >> rectWidth;
   cout << "What is the length? ";
   cin >> rectLength;
   // Store the width and length of the rectangle
   // in the box object.
   box.setWidth(rectWidth);
   box.setLength(rectLength);
   // Display the rectangle's data.
   cout << "Here is the rectangle's data:\n";
   cout << "Width: " << box.getWidth() << endl;
   cout << "Length: " << box.getLength() << endl;
   cout << "Area: " << box.getArea() << endl;
   return 0;
}
```

Pointer to an Object and Dynamically Allocating an Object

- Can define a pointer to an object:

```
Rectangle *rPtr = nullptr;
```

- Can access public members via pointer:

```
rPtr = &otherRectangle;
rPtr->setLength(12.5);
cout << rPtr->getLength() << endl;
```

- We can also use a pointer to dynamically allocate an object

```
Rectangle *rPtr = nullptr;  // define pointer
rPtr = new Rectangle;  // allocate object
rPtr->setWidth(10.0);  // store values
rPtr->setLength(15.0);
delete rPtr;           // delete object
rPtr = nullptr;
```

Separating Specification from Implementation

- For better software engineering, it is recommended that the class declaration and implementation be kept in two separate files: declaration is a header file ".h"; while implementation in a ".cpp".
- This is known as separating the public interface (header declaration) and the implementation. Interface is defined by the designer, implementation can be supplied by others. While the interface is fixed, different vendors can provide different implementations. Furthermore, only the header files are exposed to the users, the implementation can be provided in an object file ".o" (or in a library). The source code needs not given to the users.

Separating Specification from Implementation

- Place class declaration in a header file that serves as the class specification file. Name the file *`ClassName`*`.h`, for example, `Rectangle.h, Circle.h`
- Place member function definitions in *`ClassName`*`.cpp`, for example, `Rectangle.cpp`, `Circle.cpp.` File should `#include` the class specification file
- Programs that use the class must `#include` the class specification file, and be compiled and linked with the member function definitions. Example: `TestCircle.cpp`

Separating Specification from Implementation – Example Circle Class

Instead of putting all the codes in a single file, we separate the interface and implementation by placing the codes in 3 files:

1. Circle.h: defines the public interface of the Circle class
2. Circle.cpp: provides the implementation of the Circle class
3. TestCircle.cpp: A test driver program for the Circle class

Circle
-radius:double = 1.0 -color:string = "red"
+Circle(radius:double,color:string) +getRadius():double +setRadius(radius:double):void +getColor():string +setColor(color:string):void +getArea():double

Compiling multifile programs

- In Code::Blocks:
  - Use Project
- Linux with GNU CC:

  > g++ -o TestCircle.exe TestCircle.cpp Circle.cpp

#include Guard - #pragma once

- To prevent a header file from being included more than once:
- Use an include guard:

  `#ifndef / #define / #endif`
- Or use `#pragma once`

```
#pragma once
// test.h
#ifndef TEST_H_
#define TEST_H_
………..
// Contents of the file go here
………..
#endif
```

Circle Class – Circle.h

```
/* The Circle class Header (Circle.h) */
#include <string>   // using string
using namespace std;

// Circle class declaration
class Circle {
private:   // Accessible by members of this class only
   // private data members (variables)
   double radius;
   string color;

public:    // Accessible by ALL
   // Declare prototype of member functions
   // Constructor with default values
   Circle(double radius = 1.0, string color = "red");

   // Public getters & setters for private data members
   double getRadius() const;
   void setRadius(double radius);
   string getColor() const;
   void setColor(string color);

   // Public member Function
   double getArea() const;
};
```

Circle Class – Circle.cpp

```
/* The Circle class Implementation (Circle.cpp) */
#include "Circle.h" // user-defined header in the same directory
// Constructor default values shall only be specified in the
// declaration, cannot be repeated in definition
Circle::Circle(double r, string c) {
   radius = r;
   color = c;
}
// Public getter for private data member radius
double Circle::getRadius() const {
   return radius;
}
// Public setter for private data member radius
void Circle::setRadius(double r) {
   radius = r;
}
// Public getter for private data member color
string Circle::getColor() const {
   return color;
}
// Public setter for private data member color
void Circle::setColor(string c) {
   color = c;
}
// A public member function
double Circle::getArea() const {
   return radius*radius*3.14159265;
}
```

Circle Class – TestCircle.cpp

```
/* A test driver for the Circle class (TestCircle.cpp) */
#include <iostream>
#include "Circle.h"   // using Circle class
using namespace std;

int main() {
   // Construct an instance of Circle c1
   Circle c1(1.2, "red");
   cout << "Radius=" << c1.getRadius() << " Area=" << c1.getArea()
        << " Color=" << c1.getColor() << endl;

   c1.setRadius(2.1);   // Change radius and color of c1
   c1.setColor("blue");
   cout << "Radius=" << c1.getRadius() << " Area=" << c1.getArea()
        << " Color=" << c1.getColor() << endl;

   // Construct another instance using the default constructor
   Circle c2;
   cout << "Radius=" << c2.getRadius() << " Area=" << c2.getArea()
        << " Color=" << c2.getColor()  << endl;
   return 0;
}
```

Inline Member Functions

- Member functions can be defined
  - inline: in class declaration
  - after the class declaration
- Inline appropriate for short function bodies:

```
int getWidth() const
    { return width; }
```

- Tradeoffs – Inline vs. Regular Member Functions:
  - Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.
  - Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence **faster** execution

Rectangle Class with Inline Member Functions

```
// Specification file for the Rectangle class
// This version uses some inline member functions.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);

      double getWidth() const
         { return width; }

     double getLength() const
         { return length; }

     double getArea() const
         { return width * length; }
};
#endif
```

Constructors

- A *constructor* is a special function that has the *function name same as the class name*
- A constructor function is different from an ordinary function in the following aspects:
  - The name of the constructor is the same as the class name.
  - Purpose is to construct an object
  - Member function that is automatically called when an object is created
  - Constructor has no return type
  - Constructors are not inherited

In-Place Initialization

- In C++11 or later, you can initialize a member variable in its declaration statement, just as you can with a regular variable.
- This is known as in-place initialization. Example:

```
class Rectangle
{
  private:
    double width = 0.0;
    double length = 0.0;
  public:
    Public member functions appear here…
};
```

Default Constructors

- A default constructor is a constructor that takes no arguments.
- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.

```
ClassName::ClassName() { }
```

- A simple instantiation of a class (with no arguments) calls the default constructor: `Rectangle r;`
- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

```
Rectangle(double = 0, double = 0);
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

Passing Arguments to Constructors

- To create a constructor that takes arguments:
  - indicate parameters in prototype:
    `Rectangle(double, double);`
  - Use parameters in the definition:

```
Rectangle::Rectangle(double w, double len){
   width = w;
   length = len;
}
```

- You can pass arguments to the constructor when you create an object: `Rectangle r(10, 5);`
- When all of a class's constructors require arguments, then the class has NO default constructor. In this case, you must pass the required arguments to the constructor when creating an object.

Overloading Constructors

- A class can have more than one constructor
- Overloaded constructors in a class must have different parameter lists:

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```

Constructor Delegation

- Sometimes a class will have multiple constructors that perform a similar set of steps. For example, look at the following `Contact` class:
- Both constructors perform a similar operation: They assign values to the name, email, and phone member variables.
- The default constructor assigns empty strings to the members, and the parameterized constructor assigns specified values to the members.

```
class Contact
{
private:
    string name;
    string email;
    string phone;
public:
    // Constructor #1 (default)
    Contact()
    { name = "";
      email = "";
      phone = "";
    }

    // Constructor #2
    Contact(string n, string e, string p)
    { name = n;
      email = e;
      phone = p;
    }
    Other member functions follow...
};
```

Constructor Delegation

- In C++ 11, it is possible for one constructor to call another constructor in the same class.
- This is known as *constructor delegation*.

```
class Contact
{
private:
   string name;
   string email;
   string phone;
public:
   // Constructor #1 (default)
   Contact() : Contact("", "", "")
   { }

   // Constructor #2
   Contact(string n, string e, string p)
   { name = n;
     email = e;
     phone = p;
   }

   Other member functions follow...
};
```

Copy Constructors

- A copy constructor constructs a new object by copying an existing object of the same type. In other words, a copy constructor takes an argument, which is an object of the same class.
- If you do not define a copy constructor, the compiler provides a default which copies all the data members of the given object. Example:

```
Circle c4(7.8, "blue");
cout << "Radius=" << c4.getRadius() << " Area=" << c4.getArea()
     << " Color=" << c4.getColor() << endl;
                // Radius=7.8 Area=191.135 Color=blue

// Construct a new object by copying an existing object
// via the so-called default copy constructor
Circle c5(c4);
cout << "Radius=" << c5.getRadius() << " Area=" << c5.getArea()
     << " Color=" << c5.getColor() << endl;
                // Radius=7.8 Area=191.135 Color=blue
```

Destructors

- Member function automatically called when an object is destroyed
- Destructor name is ~classname, *e.g.*, `~Rectangle`
- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it
- If you do not define a destructor, the compiler provides a default, which does nothing

```
class MyClass {
public:
// The default destructor that does nothing
   ~MyClass() { }
......
}
```

Only One Default Constructor and One Destructor

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters

```
Square();
Square(int = 0);  // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class

More features

- Constructors, Destructors, and Dynamically Allocated Objects:
  - When an object is dynamically allocated with the new operator, its constructor executes:
    ```
    Rectangle *r = new Rectangle(10, 20);
    ```
  - When the object is destroyed, its destructor executes:
    ```
    delete r;
    ```
- Member Function Overloading:
  - Non-constructor member functions can also be overloaded:
    ```
    void setCost(double);
    void setCost(char *);
    ```
  - Must have unique parameter lists as for constructors

Using Private Member Functions

- A `private` member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class
- Example: `createDescription` function in **`ContactInfo.h`**

Using Private Member Functions

- A `private` member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class
- Example: `createDescription` function in **`ContactInfo.h`**

Constructor Example (Rectangle.h)

```
// Specification file for the Rectangle class
// This version has a constructor.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
      Rectangle();              // Constructor
      void setWidth(double);
      void setLength(double);

      double getWidth() const
         { return width; }
      double getLength() const
         { return length; }
      double getArea() const
         { return width * length; }
};
#endif
```

Constructor Example (Rectangle.cpp)

```
// Implementation file for the Rectangle class. This version has a constructor.
#include "Rectangle.h"   // Needed for the Rectangle class
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

// The constructor initializes width and length to 0.0.     *
Rectangle::Rectangle() {
   width = 0.0;  length = 0.0;
}
// setWidth sets the value of the member variable width.    *
void Rectangle::setWidth(double w) {
   if (w >= 0)
      width = w;
   else {
      cout << "Invalid width\n";
      exit(EXIT_FAILURE);
   }
}
// setLength sets the value of the member variable length.  *
void Rectangle::setLength(double len) {
   if (len >= 0)
      length = len;
   else {
      cout << "Invalid length\n";
      exit(EXIT_FAILURE);
   }
}
```

Constructor Example (ProgRect.cpp)

```
// This program uses the Rectangle class's constructor.
#include <iostream>
#include "Rectangle.h" // Needed for Rectangle class
using namespace std;

int main()
{
   Rectangle box; // Define an instance of the Rectangle class

   // Display the rectangle's data.
   cout << "Here is the rectangle's data:\n";
   cout << "Width: " << box.getWidth() << endl;
   cout << "Length: " << box.getLength() << endl;
   cout << "Area: " << box.getArea() << endl;
   return 0;
}
```

Example (InventoryItem.h)

```
// InventoryItem.h: This class has overloaded constructors
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <string>
using namespace std;

class InventoryItem {
private:
   string description; // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
public:
   // Constructor #1 (default constructor)
   InventoryItem() { // Initialize description, cost, and units.
        description = "";
        cost = 0.0;
        units = 0; }
   // Constructor #2
   InventoryItem(string desc) { // Assign the value to description.
        description = desc;
        cost = 0.0; // Initialize cost and units.
        units = 0; }
```

Example (InventoryItem.h)

```
    // Constructor #3
    InventoryItem(string desc, double c, int u) {
         description = desc; // Assign values to desc., cost, and units.
         cost = c;
         units = u; }
    // Destructor
    ~InventoryItem() {
         delete description;}

    // Mutator functions
    void setDescription(string d) {
         description = d; }
    void setCost(double c) {
         cost = c; }
    void setUnits(int u){ units = u; }

    // Accessor functions
    string getDescription() const { return description; }

    double getCost() const { return cost; }

    int getUnits() const { return units; }
};
#endif
```

Example (InventoryItem.cpp)

```
// This program demonstrates a class with overloaded constructors.
#include <iostream>
#include <iomanip>
#include "InventoryItem.h"

int main() {
      InventoryItem item1; // Create an InventoryItem obj. with const. #1.
      item1.setDescription("Hammer");   // Set the description
      item1.setCost(6.95);             // Set the cost
      item1.setUnits(12);              // Set the units
      InventoryItem item2("Pliers"); // Create InventoryItem obj. const. 2
      InventoryItem item3("Wrench", 8.75, 20); // InventoryItem const 3
      cout << "The following items are in inventory:\n";
      cout << setprecision(2) << fixed << showpoint;
      // Display the data for items 1, 2, 3
      cout << "Description: " << item1.getDescription() << endl;
      cout << "Cost: $" << item1.getCost() << endl;
      cout << "Units on Hand: " << item1.getUnits() << endl << endl;
      cout << "Description: " << item2.getDescription() << endl;
      cout << "Cost: $" << item2.getCost() << endl;
      cout << "Units on Hand: " << item2.getUnits() << endl << endl;
      cout << "Description: " << item3.getDescription() << endl;
      cout << "Cost: $" << item3.getCost() << endl;
      cout << "Units on Hand: " << item3.getUnits() << endl;
      return 0;
}
```

More Examples

Time
-hour:int = 0 -minute:int = 0 -second:int = 0
+Time(h:int, m:int, s:int) +getHour():int +getMinute():int +getSecond():int +setHour(h:int):void +setMinute(m:int):void +setSecond(s:int):void +setTime(h:int, m:int, s:int) +print():void +nextSecond():void

Note on +print(): hh:mm:ss (e.g., 00:01:59)

Point
-x:int = 0 -y:int = 0
+Point(x:int,y:int) +getX():int +setX(x:int):void +getY():int +setY(y:int):void +setXY(x:int,y:int):void +getMagnitude():double +getArgument():double +print():void

Note on +print(): (x,y)

More Examples

Account
-accountNumber:int -balance:double = 0.0
+Account(accountNumber:int, balance:double) +getAccountNumber():int +getBalance():int +setBalance(balance:double):void +credit(amount:double):void +debit(amount:double):void +print():void

Note (on print()): A/C no: xxx Balance=$xxx

Ball
-x:double = 0.0 -y:double = 0.0 -xSpeed:double = 0.0 -ySpeed:double = 0.0
+Ball(x:double, y:double xSpeed:double, ySpeed:double) +getX():double +setX(x:double):void +getY():double +setY(y:double):void +getXSpeed():double +setXSpeed(xSpeed:double):void +getYSpeed():double +setYSpeed(ySpeed:double):void +setXY(x:double, y:double):void +setXYSpeed(xSpeed:double, ySpeed:double):void +move():void +print():void

More Examples

Author

```
-name:string
-email:string
-gender:char
```

```
+Author(name:string, email:string,
   gender:char)
+getName():string
+getEmail():string
+setEmail(email:string):void
+getGender():char
+print():void
```

Contain a '@'.

'm', 'f', or 'u' for unknown

"name (gender) at email"

aggregation (or *has-a*) association relationship

Book

```
-name:string
-author:Author
-price:double
-qtyInStock:int = 0
```

```
+Book(name:string, author:Author,
  price:double, qtyInStock:int)
+getName():string
+getAuthor():Author
+getPrice():double
+setPrice(price:double):void
+getQtyInStock():int
+setQtyInStock(qtyInStock:int):void
+print():void
+getAuthorName():string
```

1

Author

```
-name:string
-email:string
-gender:char
```

More Examples

Complex
-real:double = 0.0 -imag:double = 0.0
+Complex(real:double, imag:double) +getReal():double +setReal(real:double):void +getImag():double +setImag(imag:double):void +setValue(real:double, imag:double):void +print():void +isReal():bool +isImaginary():bool +addInto(another:MyComplex&):MyComplex& +addInto(real:double, imag:double):MyComplex& +addReturnNew(another:MyComplex&):MyComplex +addReturnNew(real:double, imag:double) :MyComplex

Date
-year:int -month:int -day:int -STR_MONTHS:string[] = {"Jan","Feb","Mar","Apr","May","Jun", "Jul","Aug","Sep","Oct","Nov","Dec"} -STR_DAY:string[] = {"Sunday","Monday","Tuesday","Wednesday", "Thursday","Friday","Saturday"} -DAYS_IN_MONTH:int[] = {31,28,31,30,31,30,31,31,30,31,30,31}
+isLeapYear(year:int):bool +isValidDate(year:int, month:int, day:int):bool +getDayOfWeek(year:int, month:int, day:int):int +Date(year:int, month:int, day:int) +setDate(year:int, month:int, day:int):void +getYear():int +getMonth():int +getDay():int +setYear(year:int):void +setMonth(month:int):void +setDay(day:int):void +print():void +nextDay():Date& +previousDay():Date& +nextMonth():Date& +previousMonth():Date& +nextYear():Date& +previousYear():Date&

Arrays of Objects

- Objects can be the elements of an array:

```
InventoryItem inventory[40];
```

- Default constructor for object is used when array is defined
- Must use initializer list to invoke constructor that takes arguments:

```
InventoryItem inventory[3]={"Hammer","Wrench","Pliers"};
```

- If the constructor requires more than one argument, the initializer must take the form of a function call:

```
InventoryItem inventory[3] = { InventoryItem("Hammer", 6.95, 12),
                               InventoryItem("Wrench", 8.75, 20),
                               InventoryItem("Pliers", 3.75, 10) };
```

- It isn't necessary to call the same constructor for each object in an array:

```
InventoryItem inventory[3] = { "Hammer",
                               InventoryItem("Wrench", 8.75, 20),
                               "Pliers" };
```

Accessing Objects in an Array

- Objects in an array are referenced using subscripts
- Member functions are referenced using dot notation:

```
inventory[2].setUnits(30);
cout << inventory[2].getUnits();
```

Instance and Static Members

- instance variable: a member variable in a class. Each object has its own copy.
- `static` variable: one variable shared among all objects of a class
- `static` member function: can be used to access `static` member variable; can be called before any objects are defined

`static` member variable

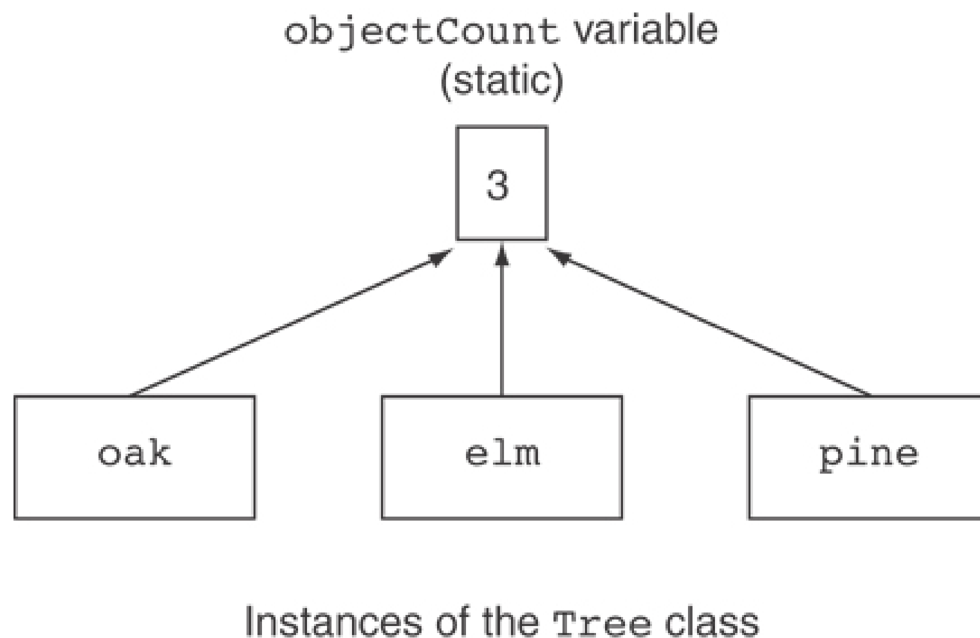
Contents of `Tree.h`

Static member declared here

```
// Tree class
class Tree
{
 private:
     static int objectCount;     // Static member variable.
 public:
    // Constructor
    Tree()
       { objectCount++; }

    // Accessor function for objectCount
    int getObjectCount() const
       { return objectCount; }
};

// Definition of the static member variable, written
// outside the class.
int Tree::objectCount = 0;
```

Static member defined here

Three Instances of the Tree Class, But Only One `objectCount` Variable



`static` member function

- Declared with `static` before return type:

```
static int getObjectCount() const
{ return objectCount; }
```

- Static member functions can only access static member data
- Can be called independent of objects:

```
int num = Tree::getObjectCount();
```

`static` member function

Modified Version of `Tree.h`

```
// Tree class
class Tree
{
  private:
    static int objectCount;    // Static member variable.
  public:
    // Constructor
    Tree()
       { objectCount++; }

   // Accessor function for objectCount
   static int getObjectCount() const
      { return objectCount; }
};


 // Definition of the static member variable, written
 // outside the class.
 int Tree::objectCount = 0;
```

Now we can call the function like this:

```
cout << "There are " << Tree::getObjectCount()
     << " objects.\n";
```

Friends of Classes

- Friend: a function or class that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with `friend` keyword in the function prototype:
  - Stand-alone function:

```
friend void setAVal(intVal&, int);
// declares setAVal function to be
// a friend of this class
```

  - Member function of another class:

```
friend void SomeClass::setNum(int num)
// setNum function from SomeClass
// class is a friend of this class
```

`friend` Class Declarations

- Class as a friend of a class:

```
class FriendClass
{
   ...
};
class NewClass
{
  public:
    friend class FriendClass; // declares
  // entire class FriendClass as a friend
  // of this class
  …
};
```

Memberwise Assignment

- Can use = to assign one object to another, or to initialize an object with an object's data
- Copies member to member. *e.g.*,

  `instance2 = instance1;` means:

  copy all member values from `instance1` and assign to the corresponding member variables of `instance2`
- Use at initialization:

```
Rectangle r2 = r1;
```

Copy Constructors

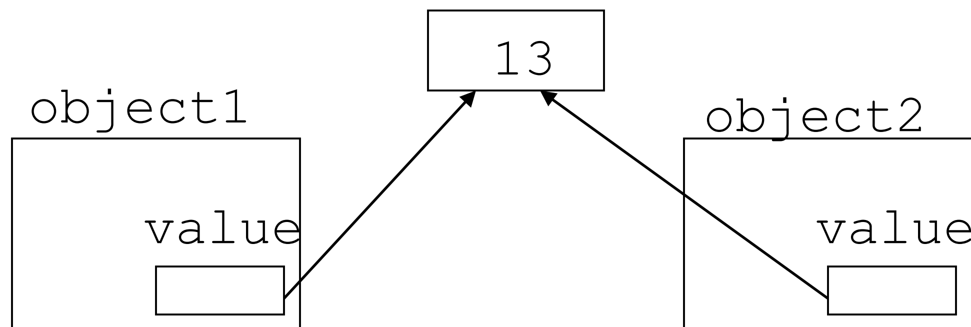
- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field
- Default copy constructor works fine in many cases
- Problem: what if object contains a pointer?

```
class SomeClass
{ public:
      SomeClass(int val = 0)
         {value=new int; *value = val;}
      int getVal();
      void setVal(int);
    private:
      int *value;
}
```

Copy Constructors

What we get using memberwise copy with objects containing dynamic memory:

```
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // also 13
```



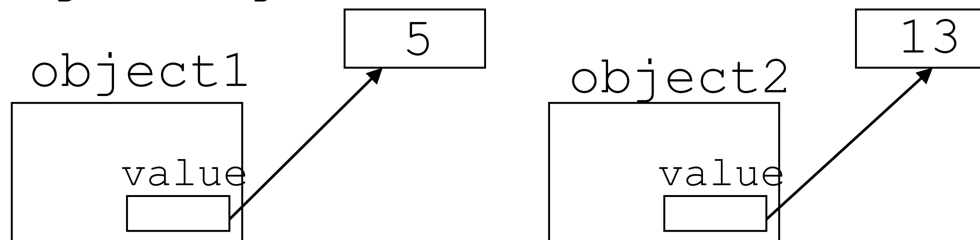
Programmer-Defined Copy Constructor

- Allows us to solve problem with objects containing pointers:

```
SomeClass::SomeClass(const SomeClass &obj)
{
    value = new int;
    *value = obj.value;
}
```

- Copy constructor takes a reference parameter to an object of the class
- Each object now points to separate dynamic memory:

```
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // still 5
```



Programmer-Defined Copy Constructor

- Since copy constructor has a reference to the object it is copying from,

```
SomeClass::SomeClass(SomeClass &obj)
```

it can modify that object.

- To prevent this from happening, make the object parameter `const`:

```
SomeClass::SomeClass
                (const SomeClass &obj)
```

StudentTestScores.h

Operator Overloading

- Operators such as =, +, and others can be redefined when used with objects of a class
- The name of the function for the overloaded operator is `operator` followed by the operator symbol, *e.g.*,
  - `operator+` to overload the + operator, and
  - `operator=` to overload the = operator
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it
- Overloaded operator function definition goes with other member functions

The `this` Pointer

- `this`: predefined pointer available to a class's member functions
- Always points to the instance (object) of the class whose function is being called
- Is passed as a hidden argument to all non-static member functions
- Example, `student1` are `StudentTestScores` object
- The following statement causes the `getStudentName` member function to operate on `student1`:

```
cout << student1.getStudentName() << endl;
```

- When `getStudentName` is operating on `student1`, the `this` pointer is pointing to `student1`.

Operator Overloading and Invoking an Overloaded Operator

- Prototype:

```
void operator=(const SomeClass &rval)
```


- Operator is called via object on left side
- Operator can be invoked as a member function:

```
object1.operator=(object2);
```

- It can also be used in more conventional manner:

```
object1 = object2;
```

Returning a Value

- Overloaded operator can return a value

```
class Point2d
{
 private:
    int x, y;

  ...
 public:
  double operator-(const point2d &right)
  { return sqrt(pow((x-right.x),2)
             + pow((y-right.y),2));
  }
};
Point2d point1(2,2), point2(4,4);
// Compute and display distance between 2 points.
cout << point2 - point1 << endl; // displays 2.82843
```

Returning a Value

- Return type the same as the left operand supports notation like:

```
object1 = object2 = object3;
```

- Function declared as follows:

```
const SomeClass operator=(const someClass &rval)
```

- In function, include as last statement:

```
return *this;
```

Notes on Overloaded Operators

- Can change meaning of an operator
- Cannot change the number of operands of the operator
- Only certain operators can be overloaded. Cannot overload the following operators:

```
?:   .   .*   :: sizeof
```

Overloading Types of Operators

- `++`, `--` operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a `bool` value
- Overloaded stream operators `>>`, `<<` must return reference to `istream`, `ostream` objects and take `istream`, `ostream` objects as parameters

Overloaded [] Operator

- Can create classes that behave like arrays, provide bounds-checking on subscripts
- Must consider constructor, destructor
- Overloaded [] returns a reference to object, not an object itself

Object Conversion

- Type of an object can be converted to another type
- Automatically done for built-in data types
- Must write an operator function to perform conversion
- To convert an `FeetInches` object to an `int`:

```
FeetInches::operator int()
                {return feet;}
```

- Assuming distance is a `FeetInches` object, allows statements like:

```
int d = distance;
```

Aggregation

- Aggregation: a class is a member of a class
- Supports the modeling of ‘has a’ relationship between classes – enclosing class ‘has a’ enclosed class
- Same notation as for structures within structures

Aggregation

```
class StudentInfo
{
   private:
          string firstName, LastName;
          string address, city, state, zip;
  ...
};
class Student
{
   private:
          StudentInfo personalData;
  ...
};
```

Aggregation

Course

- courseName : char [ ]
- instructor : Instructor
- textBook : TextBook

+ Course(name : char *, instr : &Instructor, text : &TextBook) :
+ print() : void

Instructor

- lastName : char [ ]
- firstName : char [ ]
- officeNumber : char [ ]

+ Instructor(lname : char *, fname : char *, office : char *) :
+ Instructor(obj : &Instructor) :
+ set(lname : char *, fname : char *, office : char *) : void
+ print() : void

TextBook

- title : char [ ]
- author : char [ ]
- publisher : char [ ]

+ TextBook(textTitle : char *, auth : char *, pub : char *) :
+ TextBook(obj : &TextBook) :
+ set(textTitle : char *, auth : char *, pub : char *) : void
+ print() : void

Lvalues and Rvalues

- Two types of values stored in memory during the execution of a program:
  - Values that persist beyond the statement that created them, and have names that make them accessible to other statements in the program. In C++, these values are called *lvalues*.
  - Values that are temporary, and cannot be accessed beyond the statement that created them. In C++, these values are called *rvalues*.

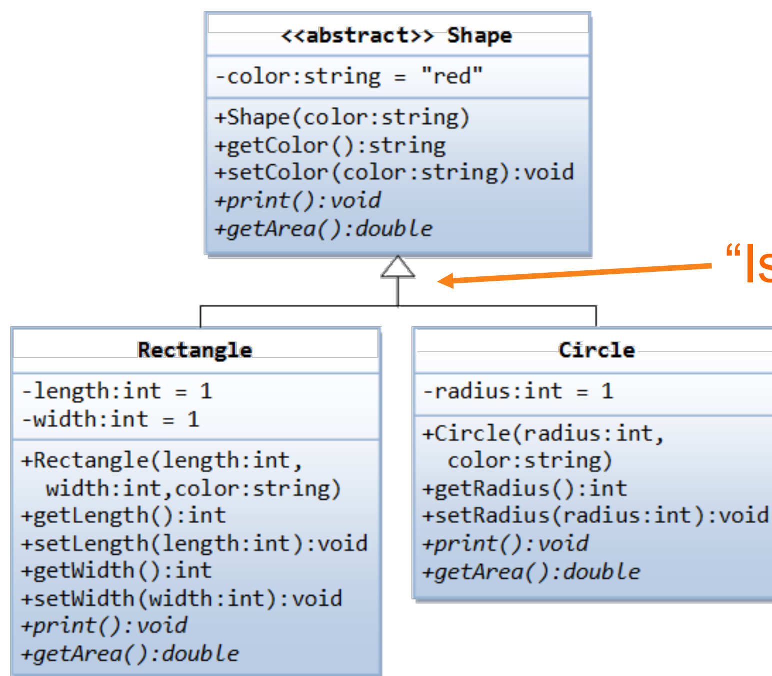
Rvalue References

- Rvalue Reference: a reference variable that can refer only to temporary objects that would otherwise have no name.
- Rvalue references are used to write move constructors and move assignment operators (otherwise known as move semantics).
- Anytime you write a class with a pointer or reference to a piece of data outside the class, you should implement move semantics.
- Move semantics increase the performance of these types of classes.

2. Inheritance, Polymorphism, and Virtual Functions

What Is Inheritance?

- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class
- Example:



Inheritance – Terminology and Notation

- Base class (or parent) – inherited from
- Derived class (or child) – inherits from the base class
- Notation:

```
class Student                // base class
{
  . . .
};
class UnderGrad : public student
{                                // derived class
  . . .
};
```

Back to the ‘is a’ Relationship

- An object of a derived class 'is a(n)' object of the base class
- Example:
  - an `UnderGrad` is a `Student`
  - a `Mammal` is an `Animal`
- A derived object has all of the characteristics of the base class

What Does a Child Have?

An object of the derived class has:

- all members defined in child class
- all members declared in parent class

An object of the derived class can use:

- all `public` members defined in child class
- all `public` members defined in parent class

Protected Members and Class Access

- `protected` member access specification: like `private`, but accessible by objects of derived class

- Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class

Class Access Specifiers

1) `public` – object of derived class can be treated as object of base class (not vice-versa)
2) `protected` – more restrictive than `public`, but allows derived classes to know details of parents
3) `private` – prevents objects of derived class from being treated as objects of base class.

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:

derived class constructor | base class constructor

```
Square::Square(int side):Rectangle(side,side)
```

derived constructor parameter | base constructor parameters

- Can also be done with inline constructors
- Must be done if base class has no default constructor

Constructor Inheritance

- In a derived class, some constructors can be inherited from the base class.

- The constructors that ***cannot*** be inherited are:
  - the default constructor
  - the copy constructor
  - the move constructor

Constructor Inheritance

- Consider the following:

```
class MyBase
{
private:
    int ival;
    double dval;
public:
    MyBase(int i)
    { ival = i; }

    MyBase(double d)
    { dval = d; }
};
```

```
class MyDerived : MyBase
{
public:
    MyDerived(int i) : MyBase(i)
    {}

    MyDerived(double d) : MyBase(d)
    {}
};
```

Constructor Inheritance

- We can rewrite the MyDerived class as:

```
class MyBase
{
private:
    int ival;
    double dval;
public:
    MyBase(int i)
    { ival = i; }

    MyBase(double d)
    { dval = d; }
};
```

```
class MyDerived : MyBase
{
    using MyBase::MyBase;
};
```

The `using` statement causes the class to inherit the base class constructors.

Redefining Base Class Functions

- Redefining function: function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class
- Not the same as overloading – with overloading, parameter lists must be different
- Objects of base class use base class version of function; objects of derived class use derived class version of function

Base Class

```
class GradedActivity
{
protected:
   char letter;            // To hold the letter grade
   double score;           // To hold the numeric score
   void determineGrade();  // Determines the letter grade
public:
   // Default constructor
   GradedActivity()
      { letter = ' '; score = 0.0; }

   // Mutator function
   void setScore(double s)
      { score = s;
        determineGrade();}

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const
      { return letter; }
};
```

Note `setScore` function

Derived Class

```
1  #ifndef CURVEDACTIVITY_H
2  #define CURVEDACTIVITY_H
3  #include "GradedActivity.h"
4
5  class CurvedActivity : public GradedActivity
6  {
7  protected:
8     double rawScore;      // Unadjusted score
9     double percentage;    // Curve percentage
10 public:
11    // Default constructor
12    CurvedActivity() : GradedActivity()
13       { rawScore = 0.0; percentage = 0.0; }
14
15    // Mutator functions
16    void setScore(double s)
17       { rawScore = s;
18         GradedActivity::setScore(rawScore * percentage); }
19
20    void setPercentage(double c)
21       { percentage = c; }
22
23    // Accessor functions
24    double getPercentage() const
25       { return percentage; }
26
27    double getRawScore() const
28       { return rawScore; }
29 };
30 #endif
```

Redefined `setScore` function

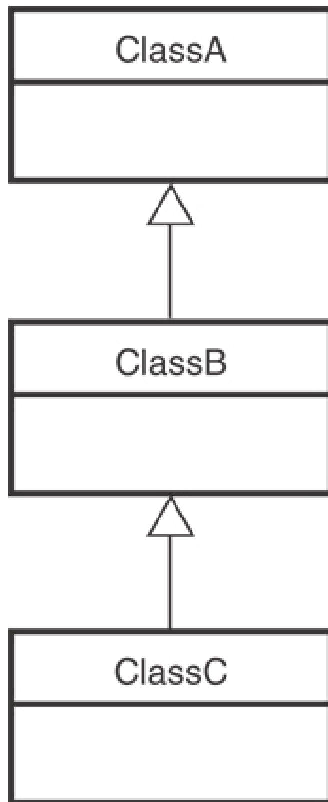
Invocation of redefined function

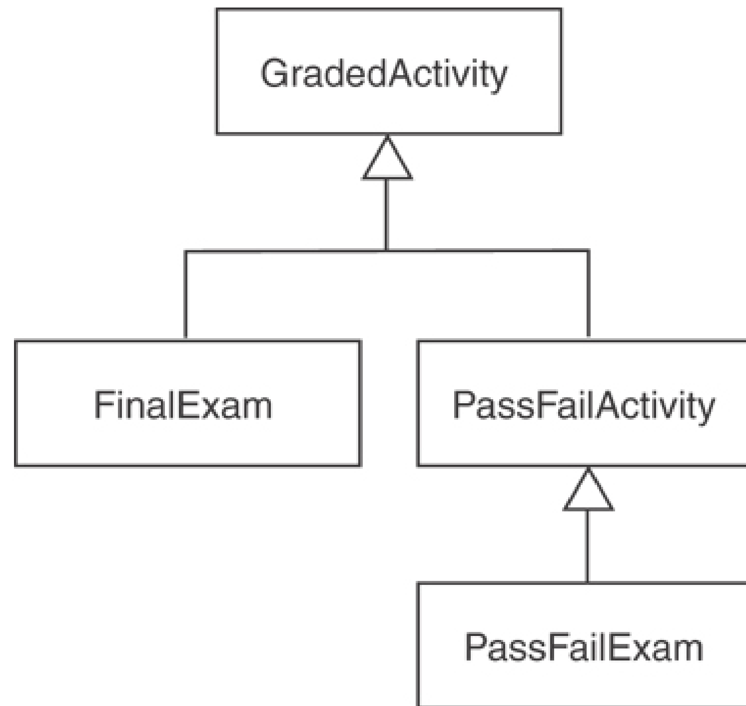
```
13      // Define a CurvedActivity object.
14      CurvedActivity exam;
15
16      // Get the unadjusted score.
17      cout << "Enter the student's raw numeric score: ";
18      cin >> numericScore;
19
20      // Get the curve percentage.
21      cout << "Enter the curve percentage for this student: ";
22      cin >> percentage;
23
24      // Send the values to the exam object.
25      exam.setPercentage(percentage);
26      exam.setScore(numericScore);
27
28      // Display the grade data.
29      cout << fixed << setprecision(2);
30      cout << "The raw score is "
31           << exam.getRawScore() << endl;
32      cout << "The curved score is "
33           << exam.getScore() << endl;
34      cout << "The curved grade is "
35           << exam.getLetterGrade() << endl;
```

Invocation `setScore` function

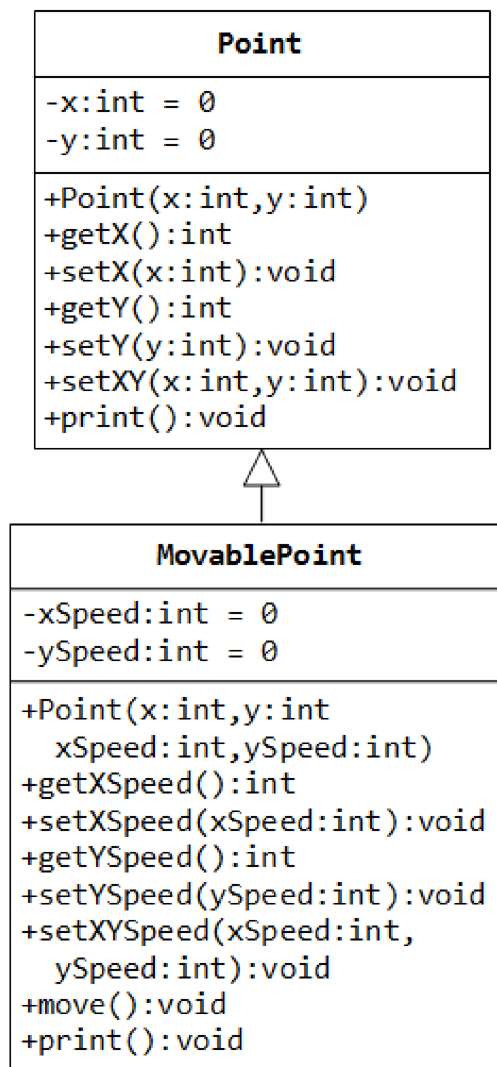
Class Hierarchies

- A base class can be derived from another base class.





Example: Superclass Point and subclass MovablePoint



Example: Shape and its Subclasses

<<abstract>> Shape
-color:string = "red"
+Shape(color:string) +getColor():string +setColor(color:string):void *+print():void* *+getArea():double*

Rectangle
-length:int = 1 -width:int = 1
+Rectangle(length:int, width:int,color:string) +getLength():int +setLength(length:int):void +getWidth():int +setWidth(width:int):void *+print():void* *+getArea():double*

Circle
-radius:int = 1
+Circle(radius:int, color:string) +getRadius():int +setRadius(radius:int):void *+print():void* *+getArea():double*

Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
- Function defined with key word `virtual`:

```
virtual void Y() {...}
```

- Supports dynamic binding: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Virtual Functions

- A virtual function is dynamically bound to calls at runtime.
- At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.
- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

```
virtual char getLetterGrade() const;
```

- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.

Updated Version of GradedActivity

```
 6  class GradedActivity
 7  {
 8  protected:
 9     double score;   // To hold the numeric score
10  public:
11     // Default constructor
12     GradedActivity()
13        { score = 0.0; }
14
15     // Constructor
16     GradedActivity(double s)
17        { score = s; }
18
19     // Mutator function
20     void setScore(double s)
21        { score = s; }
22
23     // Accessor functions
24     double getScore() const
25        { return score; }
26
27     virtual char getLetterGrade() const;
28  };
```

The function is now virtual.

The function also becomes virtual in all derived classes automatically!

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer, as demonstrated in the `displayGrade` function.

Base Class Pointers

- Can define a pointer to a *base* class object
- Can assign it the address of a *derived* class object

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);

cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

- Base class pointers and references only know about members of the base class
  - So, you can't use a base class pointer to call a derived class function
- Redefined functions in *derived* class will be ignored unless *base* class declares the function `virtual`

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.
- So, a virtual function is overridden, and a non-virtual function is redefined.

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.

C++ 11's `override` and `final` Key Words

- The `override` key word tells the compiler that the function is supposed to override a function in the base class.
- When a member function is declared with the `final` key word, it cannot be overridden in a derived class.

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:

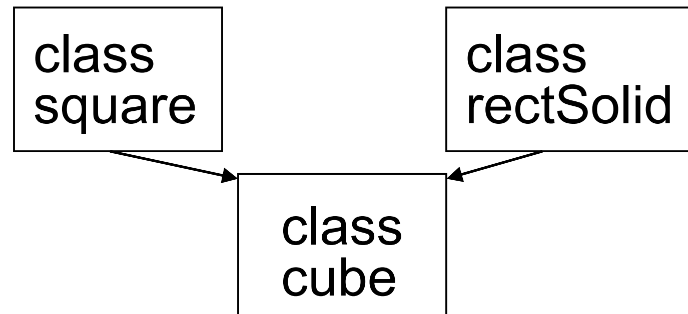
```
virtual void Y() = 0;
```

- The `= 0` indicates a pure virtual function
- Must have no function definition in the base class
- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:

```
class cube : public square, public rectSolid;
```



- Arguments can be passed to both base classes' constructors:

```
    cube::cube(int side) : square(side),
          rectSolid(side, side, side);
```

- Base class constructors are called in order given in class declaration, not in order used in class constructor

Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
  - Derived class redefines the multiply-defined function
  - Derived class invokes member function in a particular base class using scope resolution operator `::`
- Compiler errors occur if derived class uses base class function without one of these solutions

3. Exceptions and Templates

Exceptions

- Indicate that something unexpected has occurred or been detected
- Allow program to deal with the problem in a controlled manner. Can be as simple or complex as program design requires
- Terminology:
  - Exception: object or value that signals an error
  - Throw an exception: send a signal that an error has occurred
  - Catch/Handle an exception: process the exception; interpret the signal

Exceptions – Key Words

- `throw` – followed by an argument, is used to throw an exception
- `try` – followed by a block `{ }`, is used to invoke code that throws an exception
- `catch` – followed by a block `{ }`, is used to detect and process exceptions thrown in preceding `try` block. Takes a parameter that matches the type thrown.

Exceptions – Flow of Control

1) A function that throws an exception is called from within a try block
2) If the function throws an exception, the function terminates and the try block is immediately exited. A catch block to process the exception is searched for in the source code immediately following the try block.
3) If a catch block is found that matches the exception thrown, it is executed. If no catch block that matches the exception is found, the program terminates.

Exceptions – Example

```
// Example1: function that throws an exception
int totalDays(int days, int weeks) {
    if ((days < 0) || (days > 7))
      throw "invalid number of days";
// the argument to throw is the
// character string
  else
      return (7 * weeks + days);
}
// Example2: try catch
try { // block that calls function
   totDays = totalDays(days, weeks);
   cout << "Total days: " << days;
}
catch (char *msg) { // interpret exception
   cout << "Error: " << msg;
}
```

Exceptions – What Happens

1) `try` block is entered. `totalDays` function is called
2) If 1st parameter is between 0 and 7, total number of days is returned and `catch` block is skipped over (no exception thrown)
3) If exception is thrown, function and `try` block are exited, `catch` blocks are scanned for 1st one that matches the data type of the thrown exception. `catch` block executes

Exceptions

What Happens in theTry/Catch Construct

If this statement throws an exception... → `quotient = divide(num1, num2);`

... then this statement is skipped. → `cout << "The quotient is " << quotient << endl;`

If the exception is a string, the program jumps to this catch clause. → `catch (char *exceptionString)`

After the catch block is finished, the program resumes here. → `cout << "End of the program.\n";`

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
    cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

What if no exception is thrown?

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct. → `cout << "End of the program.\n";`

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
    cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

Exceptions - Notes

- Predefined functions such as `new` may throw exceptions
- The value that is thrown does not need to be used in `catch` block.
  - in this case, no name is needed in catch parameter definition
  - `catch` block parameter definition *does* need the type of exception being caught

Exception Not Caught?

- An exception will not be caught if
  - it is thrown from outside of a `try` block
  - there is no `catch` block that matches the data type of the thrown exception
- If an exception is not caught, the program will terminate

Exceptions and Objects

- An exception class can be defined in a class and thrown as an exception by a member function
- An exception class may have:
  - no members: used only to signal an error
  - members: pass error data to `catch` block
- A class can have more than one exception class

What Happens After `catch` Block?

- Once an exception is thrown, the program cannot return to throw point. The function executing `throw` terminates (does not return), other calling functions in `try` block terminate, resulting in <u>unwinding the stack</u>
- If objects were created in the `try` block and an exception is thrown, they are destroyed.

Nested `try` Blocks

- `try/catch` blocks can occur within an enclosing `try` block
- Exceptions caught at an inner level can be passed up to a `catch` block at an outer level:

```
catch ( )
{
   ...
   throw;  // pass exception up
}         // to next level
```

Function Templates

- Function template: a pattern for a function that can work with many data types
- When written, parameters are left for the data types
- When called, compiler generates code for specific data types in function call

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```


What gets generated when `times10` is called with an `int`:	What gets generated when `times10` is called with a `double`:
`int times10(int num) { return 10 * num; }`	`double times10(double num) { return 10 * num; }`

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

- Call a template function in the usual manner:

```
int ival = 3;
double dval = 2.55;
cout << times10(ival); // displays 30
cout << times10(dval); // displays 25.5
```

Function Template Notes

- Can define a template to use multiple data types:

```
template<class T1, class T2>
```

- Example:

```
template<class T1, class T2>      // T1 and T2 will be
double mpg(T1 miles, T2 gallons) // replaced in the
{                                // called function
 return miles / gallons          // with the data
}                                // types of the
                                 // arguments
```

Function Template Notes

- Function templates can be overloaded Each template must have a unique parameter list

```
template <class T>
T sumAll(T num) ...
template <class T1, class T2>
T1 sumall(T1 num1, T2 num2) ...
```

- All data types specified in template prefix must be used in template definition
- Function calls must pass parameters for all data types specified in the template prefix
- Like regular functions, function templates must be defined before being called

Function Template Notes

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition

Where to Start When Defining Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop function using usual data types first, then convert to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template

Class Templates

- Classes can also be represented by templates. When a class object is created, type information is supplied to define the type of data members of the class.
- Unlike functions, classes are instantiated by supplying the type name (`int`, `double`, `string`, etc.) at object definition

Class Template Example

```
template <class T>
class grade
{
    private:
        T score;
    public:
        grade(T);
        void setGrade(T);
        T getGrade()
};
```

Class Template Example

- Pass type information to class template when defining objects:

```
grade<int> testList[20];
grade<double> quizList[20];
```

- Use as ordinary objects once defined

Class Templates and Inheritance

- Class templates can inherit from other class templates:

```
template <class T>
class Rectangle
  { ... };
template <class T>
class Square : public Rectangle<T>
  { ... };
```

- Must use type parameter `T` everywhere base class name is used in derived class

4. The Standard Template Library

The Standard Template Library

- **The Standard Template Library (STL)**: an extensive library of generic templates for classes and functions.
- Categories of Templates:
  - **Containers**: Class templates for objects that store and organize data
  - **Iterators**: Class templates for objects that behave like pointers, and are used to access the individual data elements in a container
  - **Algorithms**: Function templates that perform various operations on elements of containers
  - **Function objects:** are objects that act like functions

The Standard Template Library headers

- <vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset>, <forward_list> (C++11), <unordered_map> (C++11), <unordered_set> (C++11), <array> (C++11): Containers data structures template classes.
- <iterator>: Iterator for transversing the elements in a container.
- <algorithm>, <numeric>, <functional>, <utility>: Algorithm and function objects.
- <initializer_list> (C++11), <memroy> (C++11).

Containers

- **Sequence Containers:** Stores data sequentially in memory
  - vector: dynamically resizable array. Support fast insertion and deletion at back; and direct access to its elements.
  - deque: double-ended queue. Support fast insertion and deletion at front and back; and direct access to its elements.
  - list: double-linked list. Support fast insertion and deletion anywhere in the list; and direct access to its elements.
- **Associative Containers:** nonlinear data structures storing key-value pairs
  - set: No duplicate element. Support fast lookup.
  - multiset: Duplicate element allowed. Support fast lookup.
  - map: One-to-one mapping (associative array) with no duplicate. Support fast key lookup.
  - multimap: One-to-many mapping, with duplicates allowed. Support fast key lookup.

Containers

- **Container Adapter Classes**
  - Stack: Last-in-first-out (LIFO) queue, adapted from deque (default), or vector, or list. Support operations back, push_back, pop_back.
  - queue: First-in-first-out (FIFO) queue, adapted from deque (default), or list. Support operations front, back, push_back, pop_front.
  - priority_queue: highest priority element at front of the queue. adapted from vector (default) or deque. Support operations front, push_back, pop_front.

STL Header Files

Header File	Classes
<array>	array
<deque>	deque
<forward_list>	forward_list
<list>	list
<map>	map, multimap
<queue>	queue, priority_queue
<set>	set, multiset
<stack>	stack
<unordered_map>	unordered_map, unordered_multimap
<unordered_set>	unordered_set, unordered_multiset
<vector>	vector

The `array` Class Template

- An `array` object works very much like a regular array
- A fixed-size container that holds elements of the same data type.
- array objects have a size() member function that returns the number of elements contained in the object.
- The `array` class is declared in the `<array>` header file.
- When defining an `array` object, you specify the data type of its elements, and the number of elements.
- Examples:

```
array<int, 5> numbers;
array<string, 4> names;
```

The array Class Template

- Initializing an `array` object:

```
array<int, 5> numbers = {1, 2, 3, 4, 5};

array<string, 4> names = {"Jamie", "Ashley", "Doug",
                          "Claire"};
```

- The `array` class overloads the [ ] operator.
- You can use the [ ] operator to access elements using a subscript, just as you would with a regular array.
- The [ ] operator does not perform bounds checking. Be careful not to use a subscript that is out of bounds.

Iterators

- An iterator behaves like a generic pointer, which can be used to reference (point-to) individual element of a generic container; and transverse through elements of a container.
- Five categories of iterators:

Iterator Category	Description
Forward	Can only move forward in a container (uses the ++ operator).
Bidirectional	Can move forward or backward in a container (uses the ++ and -- operators).
Random access	Can move forward and backward, and can jump to a specific data element in a container.
Input	Can be used with an input stream to read data from an input device or a file.
Output	Can be used with an output stream to write data to an output device or a file.

Similarities between Pointers and Iterators

	Pointers	Iterators
Use the * and -> operators to dereference	Yes	Yes
Use the = operator to assign to an element	Yes	Yes
Use the == and != operators to compare	Yes	Yes
Use the ++ operator to increment	Yes	Yes
Use the -- operator to decrement	Yes	Yes (bidirectional and random-access iterators)
Use the + operator to move forward a specific number of elements	Yes	Yes
Use the - operator to move backward a specific number of elements	Yes	Yes Yes (bidirectional and random-access iterators)

Iterators

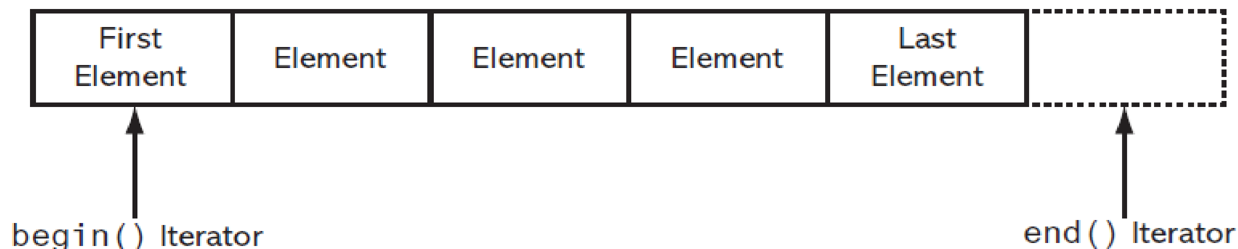
- To define an iterator, you must know what type of container you will be using it with.
- The general format of an iterator definition:
  `containerType``::iterator` *`iteratorName`*`;`
  Where *`containerType`* is the STL container type, and *`iteratorName`* is the name of the iterator variable that you are defining.
- For example, suppose we have defined an `array` object, as follows:
  ```
  array<string, 3> names = {"Sarah", "William", "Alfredo"};
  ```
  We can define an iterator that is compatible with the array object as follows:
  ```
  array<string, 3>::iterator it;
  ```
  This defines an iterator named `it`. The iterator can be used with an `array<string, 3>` object.

Iterators

- All of the STL containers have a `begin()` member function that returns an iterator pointing to the container's first element.

```
// Define an array object.
array<string, 3> names = {"Sarah", "William", "Alfredo"};

// Define an iterator for the array object.
array<string, 3>::iterator it;

// Make the iterator point to the array object's first element.
it = names.begin();

// Display the element that the iterator points to.
cout << *it << endl;
```

- All of the STL containers have a `end()` member function that returns an iterator pointing to the position *after* the container's last element.



Iterators

- You typically use the `end()` member function to know when you have reached the end of a container.

```
// Define an array object.
array<string, 3> names = {"Sarah", "William", "Alfredo"};

// Define an iterator for the array object.
array<string, 3>::iterator it;

// Make the iterator point to the array object's first element.
it = names.begin();

// Display the array object's contents.
while (it != names.end())
{
    cout << *it << endl;
    it++;
}
```

- You can use the `auto` keyword to simplify the definition of an iterator. Example:

```
array<string, 3> names = {"Sarah", "William", "Alfredo"};
auto it = names.begin();
```

Iterators

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

int main()
{
    const int SIZE = 3;

    // Store some names in an array object.
    array<string, SIZE> names = {"Sarah", "William", "Alfredo"};

    // Create an iterator for the array object.
    array<string, SIZE>::iterator it;

    // Display the names.
    cout << "Here are the names:\n";
    for (it = names.begin(); it != names.end(); it++)
        cout << *it << endl;

    return 0;
}
```

Iterators

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

int main()
{
    const int SIZE = 4;

    // Store some names in an array object.
    array<string, SIZE> names = {"Jamie", "Ashley", "Doug", "Claire"};

    // Display the names.
    cout << "Here are the names:\n";
    for (auto it = names.begin(); it != names.end(); it++)
        cout << *it << endl;

    return 0;
}
```

Mutable Iterators

- An iterator of the `iterator` type gives you read/write access to the element to which the iterator points.
- This is commonly known as a mutable iterator.

```
// Define an array object.
array<int, 5> numbers = {1, 2, 3, 4, 5};

// Define an iterator for the array object.
array<int, 5>::iterator it;

// Make the iterator point to the array object's first element.
it = numbers.begin();

// Use the iterator to change the element.
*it = 99;
```

Constant Iterators

- An iterator of the `const_iterator` type provides read-only access to the element to which the iterator points.
- The STL containers provide a `cbegin()` member function and a `cend()` member function.
  - The `cbegin()` member function returns a `const_iterator` pointing to the first element in a container.
  - The `cend()` member function returns a `const_iterator` pointing to the end of the container.
  - When working with `const_iterators`, simply use the container class's `cbegin()` and `cend()` member functions instead of the `begin()` and `end()` member functions.

Reverse Iterators

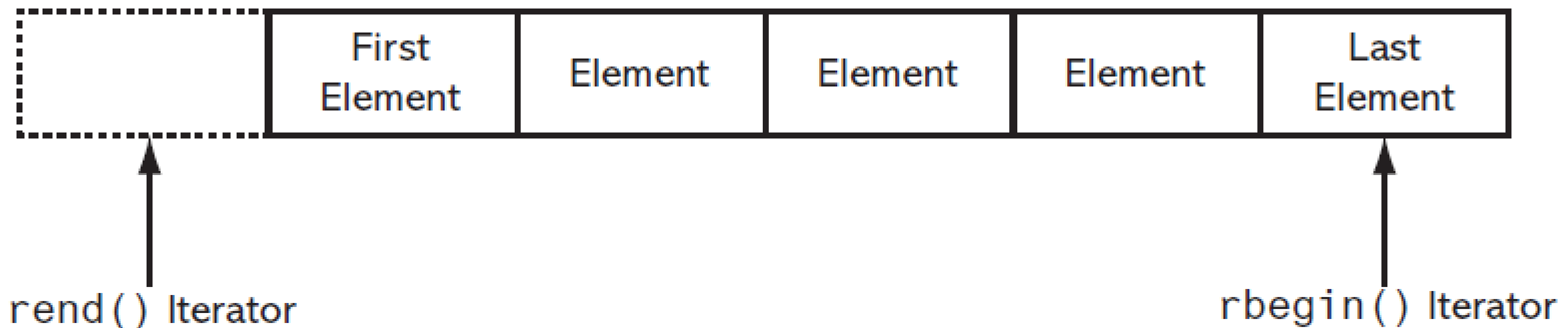
- A *reverse iterator* works in reverse, allowing you to iterate backward over the elements in a container.
- With a reverse iterator, the last element in a container is considered the first element, and the first element is considered the last element.
- The ++ operator moves a reverse iterator backward, and the –– operator moves a reverse iterator forward.

Reverse Iterators

- The following STL containers support reverse iterators:
  - `array`
  - `deque`
  - `list`
  - `map`
  - `multimap`
  - `multiset`
  - `set`
  - `vector`
- All of these classes provide an `rbegin()` member function and an `rend()` member function.

Reverse Iterators

- The `rbegin()` member function returns a reverse iterator pointing to the last element in a container.
- The `rend()` member function returns an iterator pointing to the position *before* the first element.



Reverse Iterators

- To create a reverse iterator, define it as `reverse_iterator`

```
// Define an array object.
array<int, 5> numbers = {1, 2, 3, 4, 5};

// Define a reverse iterator for the array object.
array<int, 5>::reverse_iterator it;

// Display the elements in reverse order.
for (it = numbers.rbegin(); it != numbers.rend(); it++)
   cout << *it << endl;
```

The vector Class

- A `vector` is a sequence container that works like an array, but is dynamic in size.
- Overloaded `[ ]` operator provides access to existing elements
- The `vector` class is declared in the `<vector>` header file.

vector Class Constructors

Default Constructor	`vector<dataType> name;` Creates an empty `vector`.
Fill Constructor	`vector<dataType> name(size);` Creates a `vector` of *`size`* elements. If the elements are objects, they are initialized via their default constructor. Otherwise, initialized with 0.
Fill Constructor	`vector<dataType> name(size, value);` Creates a `vector` of *`size`* elements, each initialized with *`value`*.
Range Constructor	`vector<dataType> name(iterator1, iterator2);` Creates a `vector` that is initialized with a range of values from another container. *`iterator1`* marks the beginning of the range and *`iterator2`* marks the end.
Copy Constructor	`vector<dataType> name(vector2);` Creates a `vector` that is a copy of *`vector2`*.

vector Class Example

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

int main()
{
    const int SIZE = 10;

    // Define a vector to hold 10 int values.
    vector<int> numbers(SIZE);

    // Store the values 0 through 9 in the vector.
    for (int index = 0; index < numbers.size(); index++)
       numbers[index] = index;

    // Display the vector elements.
    for (auto element : numbers)
       cout << element << " ";
    cout << endl;

    return 0;
}
```

Subscript notation

Range-based for loop

Initializing a vector

- In C++ 11 and later, you can initialize a `vector` object:

```
vector<int> numbers = {1, 2, 3, 4, 5};
```

or

```
vector<int> numbers {1, 2, 3, 4, 5};
```

Adding New Elements to a `vector`

- The `push_back` member function adds a new element to the end of a `vector`:

```
vector<int> numbers;
numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);
```

Accessing Elements with the `at()` Member Function

- You can use the `at()` member function to retrieve a `vector` element by its index with bounds checking:

```
vector<string> names = {"Joe", "Karen", "Lisa"};
cout << names.at(0) << endl;
cout << names.at(1) << endl;
cout << names.at(2) << endl;
cout << names.at(3) << endl; // Throws an exception
```

Throws an out_of_bounds exception when given an invalid index

Using an Iterator With a `vector`

- `vector`s have `begin()` and `end()` member functions that return iterators pointing to the beginning and end of the container:

```
// Create a vector containing names.
vector<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
vector<string>::iterator it;

// Use the iterator to display each element in the vector.
for (it = names.begin(); it != names.end(); it++)
{
    cout << *it << endl;
}
```

Defines an iterator that is compatible with a `vector<string>` object (annotation for `vector<string>::iterator it;`)

Displays the item that the iterator points to (annotation for `cout << *it << endl;`)

Using an Iterator With a vector

- The `begin()` and `end()` member functions return a random-access iterator of the `iterator` type

- The `cbegin()` and `cend()` member functions return a random-access iterator of the `const_iterator` type

- The `rbegin()` and `rend()` member functions return a reverse iterator of the `reverse_iterator` type

- The `crbegin()` and `crend()` member functions return a reverse iterator of the `const_reverse_iterator` type

Inserting Elements with the `insert()` Member Function

- You can use the `insert()` member function, along with an iterator, to insert an element at a specific position.
- General format:

```
vectorName.insert(it, value);
```

Inserting Elements Example

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

int main()
{
     // Define a vector with 5 int values.
     vector<int> numbers = {1, 2, 3, 4, 5};

     // Define an iterator pointing to the second element.
     auto it = numbers.begin() + 1;

     // Insert a new element with the value 99.
     numbers.insert(it, 99);

     // Display the vector elements.
     for (auto element : numbers)
        cout << element << " ";
     cout << endl;

     return 0;
}
```

Overloaded Versions of the `insert()` Member Function

`insert(it, value)`	Inserts *value* just before the element pointed to by *it*. The function returns an iterator pointing to the newly inserted element.
`insert(it, n, value)`	Inserts *n* elements just before the element pointed to by *it*. Each of the new elements will be initialized with *value*. The function returns an iterator pointing to the first element of the newly inserted elements.
`insert(iterator1, iterator2, iterator3)`	Inserts a range of new elements. *iterator1* points to an existing element in the container. The range of new elements will be inserted before the element pointed to by *iterator1*. *iterator2* and *iterator3* mark the beginning and end of a range of values that will be inserted. (The element pointed to by *iterator3* will not be included in the range.) The function returns an iterator pointing to the first element of the newly inserted range.

Storing Objects Of Your Own Classes in a `vector`

- STL containers are especially useful for storing objects of your own classes.
- Consider this `Product` class:

```
#ifndef PRODUCT_H
#define PRODUCT_H
#include <string>
using namespace std;

class Product
{
private:
    string name;
    int units;
public:
    Product(string n, int u)
    {   name = n;
        units = u; }

    void setName(string n)
    {   name = n; }

    void setUnits(int u)
    {   units = u; }

    string getName() const
    {   return name; }

    int getUnits() const
    {   return units; }
};
#endif
```

Storing Objects Of Your Own Classes in a `vector`

```
#include <iostream>
#include <vector>
#include "Product.h"
using namespace std;

int main()
{
    // Create a vector of Product objects.
    vector<Product> products =
    {
        Product("T-Shirt", 20),
        Product("Calendar", 25),
        Product("Coffee Mug", 30)
    };

    // Display the vector elements.
    for (auto element : products)
    {
        cout << "Product: " << element.getName() << endl
             << "Units: " << element.getUnits() << endl;
    }

    return 0;
}
```

This program initializes a `vector` with three `Product` objects.

A range-based `for` loop iterates over the `vector`.

Storing Objects Of Your Own Classes in a `vector`

```
#include <iostream>
#include <string>
#include <vector>
#include "Product.h"
using namespace std;

int main()
{
    // Create Product objects.
    Product prod1("T-Shirt", 20);
    Product prod2("Calendar", 25);
    Product prod3("Coffee Mug", 30);

    // Create a vector to hold the Products
    vector<Product> products;

    // Add the products to the vector.
    products.push_back(prod1);
    products.push_back(prod2);
    products.push_back(prod3);

    // Use an iterator to display the vector contents.
    for (auto it = products.begin(); it != products.end(); it++)
    {
        cout << "Product: " << it->getName() << endl
             << "Units: " << it->getUnits() << endl;
    }

    return 0;
}
```

This program uses the `push_back` member function to store three `Product` objects in a `vector`.

A `for` loop uses an iterator to step through the `vector`.

Inserting Container Elements With Emplacement

- Member functions such as `insert()` and `push_back()` can cause temporary objects to be created in memory while the insertion is taking place.
- This is not a problem in programs that make only a few insertions.
- However, these functions can be inefficient for making a lot of insertions.
- C++11 introduced a new family of member functions that use a technique known as *emplacement* to insert new elements.
- Emplacement avoids the creation of temporary objects in memory while a new object is being inserted into a container.
- The emplacement functions are more efficient than functions such as `insert()` and `push_back()`

Inserting Container Elements With Emplacement

- The `vector` class provides two member functions that use emplacement:
  - `emplace()` - emplaces an element at a specific location
  - `emplace_back()`- emplaces an element at the end of the vector

- With these member functions, it is not necessary to instantiate, ahead of time, the object you are going to insert.
- Instead, you pass to the emplacement function any arguments that you would normally pass to the constructor of the object you are inserting.
- The emplacement function handles the construction of the object, forwarding the arguments to its constructor.

Inserting Container Elements With Emplacement

```
#include <iostream>
#include <vector>
#include "Product.h"
using namespace std;

int main()
{
    // Create a vector to hold Products.
    vector<Product> products;

    // Add Products to the vector.
    products.emplace_back("T-Shirt", 20);
    products.emplace_back("Calendar", 25);
    products.emplace_back("Coffee Mug", 30);

    // Use an iterator to display the vector contents.
    for (auto it = products.begin(); it != products.end(); it++)
    {
        cout << "Product: " << it->getName() << endl
             << "Units: " << it->getUnits() << endl;
    }

    return 0;
}
```

Define a `vector` to hold `Product` objects

Emplace three `Product` objects at the end of the `vector`

A `for` loop uses an iterator to step through the `vector`.

Inserting Container Elements With Emplacement

```
#include <iostream>
#include <vector>
#include "Product.h"
using namespace std;

int main()
{
    // Create a vector to hold Products.
    vector<Product> products =
    {
        Product("T-Shirt", 20),
        Product("Coffee Mug", 30)
    };

    // Get an iterator to the 2nd element.
    auto it = products.begin() + 1;

    // Insert another Product into the vector.
    products.emplace(it, "Calendar", 25);

    // Display the vector contents.
    for (auto element : products)
    {
        cout << "Product: " << element.getName() << endl
             << "Units: " << element.getUnits() << endl;
    }

    return 0;
}
```

Initializes a `vector` with two `Product` objects

Gets an iterator pointing to the 2nd element

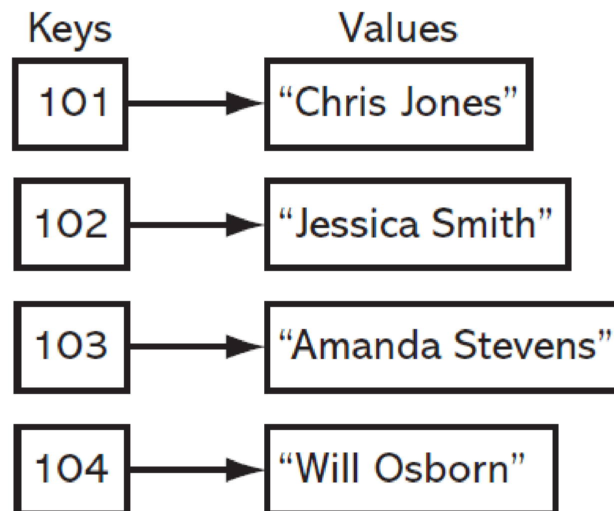
Emplaces a new `Product` object before the one pointed to by the iterator

Maps – General Concepts

- A *map* is an associative **container**.
- Each element that is stored in a map has two parts: a *key* and a *value*.
- To retrieve a specific value from a map, you use the key that is associated with that value.
- This is similar to the process of looking up a word in the dictionary, where the words are keys and the definitions are values.

Maps

- Example: a map in which employee IDs are the keys and employee names are the values.
- You use an employee's ID to look up that employee's name.



The `map` Class

- You can use the STL `map` class to store key-value pairs.
- The keys that are stored in a `map` container are unique – no duplicates.
- The `map` class is declared in the `<map>` header file.
- Example: defining a `map` container to hold employee ID numbers (as `int`s) and their corresponding employee names (as `string`s):

```
map<int, string> employees;
```

Key data type — Value data type

map Class Constructors

Default Constructor

```
map<keyDataType, valueDataType> name;
```

Creates an empty `map`.

Range Constructor

```
map<keyDataType, valueDataType>
    name(iterator1, iterator2);
```

Creates a `map` that is initialized with a range of values from another `map`. *`iterator1`* marks the beginning of the range and *`iterator2`* marks the end.

Copy Constructor

```
map<keyDataType, valueDataType> name(map2);
```

Creates a `map` that is a copy of *`map2`*.

Initializing a map

```
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};
```

- In the first element, the key is 101 and the value is "Chris Jones".
- In the second element, the key is 102 and the value is "Jessica Smith".
- In the third element, the key is 103 and the value is "Amanda Stevens".
- In the fourth element, the key is 104 and the value is "Will Osborn".

The Overloaded [ ] Operator

- You can use the `[ ]` operator to add new elements to a `map`.
- General format:

```
mapName[key] = value;
```

- This adds the key-value pair to the `map`.
- If the key already exists in the `map`, it's associated value will be changed to *`value`*.

The Overloaded [ ] Operator

```
map<int, string> employees;
employees[110] = "Beth Young";
employees[111] = "Jake Brown";
employees[112] = "Emily Davis";
```

- After this code executes, the `employees` map will contain the following elements:
  - Key = 110, Value = "Beth Young"
  - Key = 111, Value = "Jake Brown"
  - Key = 112, Value = "Emily Davis"

The `pair` Type

- Internally, the elements of a `map` are stored as instances of the `pair` type.
- `pair` is a struct that has two member variables: `first` and `second`.
- The element's key is stored in `first`, and the element's value is stored in `second`.
- The `pair` struct is declared in the `<utility>` header file. When you `#include` the `<map>` header file, `<utility>` is automatically included as well.

Inserting Elements with the `insert()` Member Function

- The `map` class provides an `insert()` member function that adds a `pair` object as an element to the `map`.
- You can use the STL function template `make_pair` to construct a `pair` object.
- The `make_pair` function template is declared in the `<utility>` header file.

Inserting Elements with the `insert()` Member Function

```
map<int, string> employees;
employees.insert(make_pair(110, "Beth Young"));
employees.insert(make_pair(111, "Jake Brown"));
employees.insert(make_pair(112, "Emily Davis"));
```

- After this code executes, the `employees` map will contain the following elements:
  - Key = 110, Value = "Beth Young"
  - Key = 111, Value = "Jake Brown"
  - Key = 112, Value = "Emily Davis"

Note: If the element that you are inserting with the `insert()` member function has the same key as an existing element, the function will *not* insert the new element.

Inserting Elements with the `emplace()` Member Function

- The `map` class also provides an `emplace()` member function that adds an element to the `map`.

```
map<int, string> employees;
employees.emplace(110, "Beth Young");
employees.emplace(111, "Jake Brown");
employees.emplace(112, "Emily Davis");
```

  - After this code executes, the `employees` map will contain the following elements:
    - Key = 110, Value = "Beth Young"
    - Key = 111, Value = "Jake Brown"
    - Key = 112, Value = "Emily Davis"

Note: If the element that you are inserting with the `emplace()` member function has the same key as an existing element, the function will *not* insert the new element.

Retrieving Elements with the `at()` Member Function

- You can use the `at()` member function to retrieve a `map` element by its key:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Retrieve a value from the map.
cout << employees.at(103) << endl;
```

Displays "Amanda Stevens"

Retrieving Elements with the `at()` Member Function

- To prevent the `at()` member function from throwing an exception (if the specified key does not exist), use the `count` member function to determine whether it exists:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Retrieve a value from the map.
if (employees.count(103))
   cout << employees.at(103) << endl;
else
   cout << "Employee not found.\n";
```

The `count()` member function returns 1 if the specified key exists, or 0 otherwise.

Deleting Elements

- You can use the `erase()` member function to retrieve a `map` element by its key:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Delete the employee with ID 102.
employees.erase(102);
```

Deletes Jessica Smith from the map

Stepping Through a `map` with the Range-Based `for` Loop

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Display each element.
for (pair<int, string> element : employees)
{
   cout << "ID: " << element.first << "\tName: "
        << element.second << endl;
}
```

Remember, each element is a `pair`.

Stepping Through a map with the Range-Based for Loop

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Display each element.
for (auto element : employees)
{
   cout << "ID: " << element.first << "\tName: "
        << element.second << endl;
}
```

auto simplifies this

Using an Iterator With a `map`

- The `begin()` and `end()` member functions return a bidirectional iterator of the `iterator` type
- The `cbegin()` and `cend()` member functions return a bidirectional iterator of the `const_iterator` type
- The `rbegin()` and `rend()` member functions return a reverse bidirectional iterator of the `reverse_iterator` type
- The `crbegin()` and `crend()` member functions return a reverse bidirectional iterator of the `const_reverse_iterator` type
- When an iterator points to a `map` element, it points to an instance of the `pair` type.
- The element has two member variables: `first` and `second`.
- The element's key is stored in `first`, and the element's value is stored in `second`.

Using an Iterator With a `map`

```
// This program demonstrates an iterator with a map.
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    // Create a map containing employee IDs and names.
    map<int, string> employees =
       { {101,"Chris Jones"}, {102,"Jessica Smith"},
         {103,"Amanda Stevens"},{104,"Will Osborn"} };

    // Create an iterator.
    map<int, string>::iterator iter;

    // Use the iterator to display each element in the map.
    for (iter = employees.begin(); iter != employees.end(); iter++)
    {
        cout << "ID: " << iter->first
             << "\tName: " << iter->second << endl;
    }

    return 0;
}
```

Storing Objects Of Your Own Classes as *Values* in a `map`

- If you want to store an object as a value in a map, there is one requirement for that object's class:

  It must have a default constructor.

- Consider the following `Contact` class…

Storing Objects Of Your Own Classes as *Values* in a `map`

```
#ifndef CONTACT_H
#define CONTACT_H
#include <string>
using namespace std;

class Contact
{
private:
    string name;
    string email;
public:
    Contact()
    {   name = "";
        email = ""; }

    Contact(string n, string em)
    {   name = n;
        email = em; }

    void setName(string n)
    {   name = n; }

    void setEmail(string em)
    {   email = em; }

    string getName() const
    {   return name; }

    string getEmail() const
    {   return email; }
};
#endif
```

Default constructor

Storing Objects Of Your Own Classes as *Values* in a `map`

```
#include <iostream>
#include <string>
#include <map>
#include "Contact.h"
using namespace std;

int main()
{
    string searchName;   // The name to search for

    // Create some Contact objects
    Contact contact1("Ashley Miller", "amiller@faber.edu");
    Contact contact2("Jacob Brown", "jbrown@gotham.edu");
    Contact contact3("Emily Ramirez", "eramirez@coolidge.edu");

    // Create a map to hold the Contact objects.
    map<string, Contact> contacts;

    // Create an iterator for the map.
    map<string, Contact>::iterator iter;

    // Add the contact objects to the map.
    contacts[contact1.getName()] = contact1;
    contacts[contact2.getName()] = contact2;
    contacts[contact3.getName()] = contact3;
```

In the map, the keys are the contact names, and the values are the `Contact` objects.

Continued…

Storing Objects Of Your Own Classes as *Values* in a `map`

```
    // Get the name to search for.
    cout << "Enter a name: ";
    getline(cin, searchName);

    // Search for the name.
    iter = contacts.find(searchName);

    // Display the results.
    if (iter != contacts.end())
    {
        cout << "Name: " << iter->second.getName() << endl;
        cout << "Email: " << iter->second.getEmail() << endl;
    }
    else
    {
        cout << "Contact not found.\n";
    }

    return 0;
}
```

Storing Objects Of Your Own Classes as *Keys* in a `map`

- If you want to store an object as a key in a map, there is one requirement for that object's class:

  It must overload the `<` operator.

- Consider the following `Customer` class…

Storing Objects Of Your Own Classes as *Keys* in a map

```
#ifndef CUSTOMER_H
#define CUSTOMER_H
#include<string>
using namespace std;

class Customer
{
private:
     int custNumber;
     string name;
public:
     Customer(int cn, string n)
     {   custNumber = cn;
         name = n; }

     void setCustNumber(int cn)
     {   custNumber = cn; }

     void setName(string n)
     {   name = n; }

     int getCustNumber() const
     {   return custNumber; }

     string getName() const
     {   return name; }

     bool operator < (const Customer &right) const
     {   bool status = false;

         if (custNumber < right.custNumber)
             status = true;

         return status; }
};
#endif
```

Storing Objects Of Your Own Classes as *Keys* in a `map`

```
#include <iostream>
#include <string>
#include <map>
#include "Customer.h"
using namespace std;

int main()
{
   // Create some Customer objects.
   Customer customer1(1001, "Sarah Scott");
   Customer customer2(1002, "Austin Hill");
   Customer customer3(1003, "Megan Cruz");

   // Create a map to hold the seat assignments.
   map<Customer, string> assignments;

   // Use the map to store the seat assignments.
   assignments[customer1] = "1A";
   assignments[customer2] = "2B";
   assignments[customer3] = "3C";

   // Display all objects in the map.
   for (auto element : assignments)
   {
      cout << element.first.getName() << "\t"
           << element.second << endl;
   }

   return 0;
}
```

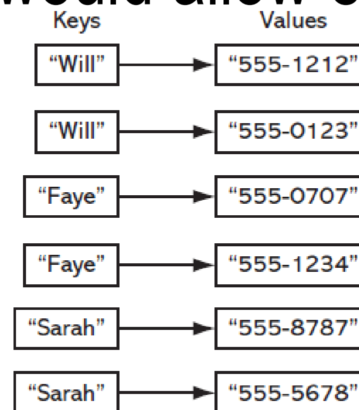
This program assigns seats in a theater to customers. The map uses Customer objects as keys, and seat numbers as values.

The `unordered_map` Class

- The `unordered_map` class is similar to the `map` class, except in two regards:
  - The keys in an `unordered_map` are not sorted
  - The `unordered_map` class has better performance
- You should use the `unordered_map` class instead of the map class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in key order
- The `unordered_map` class is declared in the `<unordered_map>` header file

The `multimap` Class

- The `multimap` class is a map that allows duplicate keys
- The `multimap` class has most of the same member functions as the map class
- The `multimap` class is declared in the `<map>` header file
- Consider a phonebook application where the key is a person's name and the value is that person's phone number.
- A `multimap` container would allow each person to have multiple phone numbers



The multimap Class

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

int main()
{
   // Define a phonebook multimap.
   multimap<string, string> phonebook =
      { {"Will", "555-1212"},  {"Will", "555-0123"},
        {"Faye", "555-0707"},  {"Faye", "555-1234"},
        {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };

   // Display the elements in the multimap.
   for (auto element : phonebook)
   {
      cout << element.first << "\t"
           << element.second << endl;
   }
   return 0;
}
```

Adding Elements to a `multimap`

- The `multimap` class does not overload the `[]` operator.
  - So, you cannot use an assignment statement to add a new element to a `multimap`.
- Instead, you will use either the `emplace()` or the `insert()` member functions.

```
multimap<string, string> phonebook;
phonebook.emplace("Will", "555-1212");
phonebook.emplace("Will", "555-0123");
phonebook.emplace("Faye", "555-0707");
phonebook.emplace("Faye", "555-1234");
phonebook.emplace("Sarah", "555-8787");
phonebook.emplace("Sarah", "555-5678");
```

Adding Elements to a `multimap`

```
multimap<string, string> phonebook;
phonebook.insert(make_pair("Will", "555-1212"));
phonebook.insert(make_pair("Will", "555-0123"));
phonebook.insert(make_pair("Faye", "555-0707"));
phonebook.insert(make_pair("Faye", "555-1234"));
phonebook.insert(make_pair("Sarah", "555-8787"));
phonebook.insert(make_pair("Sarah", "555-5678"));
```

Getting the Number of Elements With a Specified Key

- The `multimap` class's `count()` member function accepts a key as its argument, and returns the number of elements that match the specified key.

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

int main()
{
   // Define a phonebook multimap.
   multimap<string, string> phonebook =
      { {"Will", "555-1212"},  {"Will", "555-0123"},
        {"Faye", "555-0707"},  {"Faye", "555-1234"},
        {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };

   // Display the number of elements that match "Faye".
   cout << "Faye has " << phonebook.count("Faye") << " elements.\n";
   return 0;
}
```

Retrieving Elements with a Specified Key

- The `multimap` class has a `find()` member function that searches for an element with a specified key.
- The `find()` function returns an iterator to the first element matching it.
- If the element is not found, the `find()` function returns an iterator to the end of the `multimap`.
- To retrieve all elements matching a specified key, use the `equal_range` member function.
- The `equal_range` member function returns a pair object.
  - The `pair` object's `first` member is an iterator pointing to the first element that matches the specified key.
  - The `pair` object's `second` member is an iterator pointing to the position *after* the last element that matches the specified key.

Retrieving Elements with a Specified Key

```
// Define a phonebook multimap.
multimap<string, string> phonebook =
   { {"Will", "555-1212"},  {"Will", "555-0123"},
     {"Faye", "555-0707"},  {"Faye", "555-1234"},
     {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };

// Define a pair variable to receive the object that
// is returned from the equal_range member function.
pair<multimap<string, string>::iterator,
     multimap<string, string>::iterator> range;

// Define an iterator for the multimap.
multimap<string, string>::iterator iter;

// Get the range of elements that match "Faye".
range = phonebook.equal_range("Faye");

// Display all of the elements that match "Faye".
for (iter = range.first; iter != range.second; iter++)
{
   cout << iter->first << "\t" << iter->second << endl;
}
```

Deleting Elements with a Specified Key

- To delete all elements matching a specified key, use the `erase()` member function.

```
// Define a phonebook multimap.
multimap<string, string> phonebook =
   { {"Will", "555-1212"},  {"Will", "555-0123"},
     {"Faye", "555-0707"},  {"Faye", "555-1234"},
     {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };

// Delete Will's phone numbers from the multimap.
phonebook.erase("Will");
```

The `unordered_multimap` Class

- The `unordered_multimap` class is similar to the `multimap` class, except:
  - The keys in an `unordered_multimap` are not sorted
  - The `unordered_multimap` class has better performance

- You should use the `unordered_multimap` class instead of the `multimap` class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in key order

- The `unordered_multimap` class is declared in the `<unordered_multimap>` header file

Sets

- A *set* is an associative container that is similar to a mathematical set.
- You can use the STL `set` class to create a set container.
- All the elements in a `set` must be unique. No two elements can have the same value.
- The elements in a set are automatically sorted in ascending order.
- The `set` class is declared in the `<set>` header file.

The `set` Class

- You can use the STL `set` class to create a set container.
- The keys that are stored in a `map` container are unique – no duplicates.
- The `map` class is declared in the `<map>` header file.

set Class Constructors

Default Constructor	`set<`*`dataType`*`>` *`name`*`;` Creates an empty `set`.
Range Constructor	`set<`*`dataType`*`>` *`name`*`(`*`iterator1`*`,` *`iterator2`*`);` Creates a `set` that is initialized with a range of values. *`iterator1`* marks the beginning of the range and *`iterator2`* marks the end.
Copy Constructor	`set<`*`dataType`*`>` *`name`*`(`*`set2`*`);` Creates a `set` that is a copy of *`set2`*.

The set Class

- Example: defining a `set` container to hold integers:
```
set<int> numbers;
```
- Example: defining and initializing a `set` container to hold integers:
```
set<int> numbers = {1, 2, 3, 4, 5};
```
- A set cannot contain duplicate items.
- If the same value appears more than once in an initialization list, it will be added to the set only one time.
- For example, the following set will contain the values 1, 2, 3, 4, and 5:
```
set<int> numbers = {1, 1, 2, 2, 3, 4, 5, 5, 5};
```

Adding New Elements to a `set`

- The `insert()` member function adds a new element to a `set`:

```
set<int> numbers;
numbers.insert(10);
numbers.insert(20);
numbers.insert(30);
```

Stepping Through a `set` With the Range-Based `for` Loop

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Display each element.
for (string element : names)
{
   cout << element << endl;
}
```

Using an Iterator With a `set`

- The `begin()` and `end()` member functions return a bidirectional iterator of the `iterator` type
- The `cbegin()` and `cend()` member functions return a bidirectional iterator of the `const_iterator` type
- The `rbegin()` and `rend()` member functions return a reverse bidirectional iterator of the `reverse_iterator` type
- The `crbegin()` and `crend()` member functions return a reverse bidirectional iterator of the `const_reverse_iterator` type

Using an Iterator With a set

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
set<string>::iterator iter;

// Use the iterator to display each element in the set.
for (iter = names.begin(); iter != names.end(); iter++)
{
    cout << *iter << endl;
}
```

Determining Whether an Element Exists

- The `set` class's `count()` member function accepts a value as its argument, and returns 1 if that value exists in the set. The function returns 0 otherwise.

```
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};
if (names.count("Lisa"))
    cout << "Lisa was found in the set.\n";
else
    cout << "Lisa was not found.\n";
```

Retrieving an Element

- The `set` class has a `find()` member function that searches for an element with a specified value.

- The `find()` function returns an iterator to the element matching it.

- If the element is not found, the `find()` function returns an iterator to the end of the `set`.

Retrieving an Element

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
set<string>::iterator iter;

// Find "Karen".
iter = names.find("Karen");

// Display the result.
if (iter != names.end())
{
   cout << *iter << " was found.\n";
}
else
{
   cout << "Karen was not found.\n";
}
```

Storing Objects Of Your Own Classes in a `set`

- If you want to store an object in a `set`, there is one requirement for that object's class:

  It must overload the `<` operator.

- Consider the following `Customer` class…

Storing Objects Of Your Own Classes in a `set`

```
#ifndef CUSTOMER_H
#define CUSTOMER_H
#include<string>
using namespace std;

class Customer
{
private:
    int custNumber;
    string name;
public:
    Customer(int cn, string n)
    {   custNumber = cn;
        name = n; }

    void setCustNumber(int cn)
    {   custNumber = cn; }

    void setName(string n)
    {   name = n; }

    int getCustNumber() const
    {   return custNumber; }

    string getName() const
    {   return name; }

    bool operator < (const Customer &right) const
    {   bool status = false;

        if (custNumber < right.custNumber)
            status = true;

        return status; }
};
#endif
```

Storing Objects Of Your Own Classes in a set

```
#include <iostream>
#include <set>
#include "Customer.h"
using namespace std;

int main()
{
    // Create a set of Customer objects.
    set<Customer> customerset =
      { Customer(1003, "Megan Cruz"),
        Customer(1002, "Austin Hill"),
        Customer(1001, "Sarah Scott")
      };

    // Try to insert a duplicate customer number.
    customerset.emplace(1001, "Evan Smith");

    // Display the set elements
    cout << "List of customers:\n";
    for (auto element : customerset)
    {
       cout << element.getCustNumber() << " "
            << element.getName() << endl;
    }
```

Continued…

```
    // Search for customer number 1002.
    cout << "\nSearching for Customer Number 1002:\n";
    auto it = customerset.find(Customer(1002, ""));

    if (it != customerset.end())
       cout << "Found: " << it->getName() << endl;
    else
       cout << "Not found.\n";

    return 0;
}
```

The `multiset` Class

- The `mulitset` class is a set that allows duplicate items.
- The `mulitset` class has the same member functions as the `set` class.
- The `multiset` class is declared in the `<set>` header file.
- In the `set` class, the `count()` member function returns either 0 or 1. In the `multiset` class, the `count()` member function can return values greater than 1.
- In the `set` class, the `equal_range()` member function returns a range with, at most, one element. In the `multiset` class, the `equal_range()` member function can return a range with multiple elements.

The unordered_set Class

- The `unordered_set` class is similar to the `set` class, except in two regards:
  - The values in an `unordered_set` are not sorted
  - The `unordered_set` class has better performance
- You should use the `unordered_set` class instead of the `set` class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in ascending order
- The `unordered_set` class is declared in the `<unordered_set>` header file

The unordered_multiset Class

- The unordered_multiset class is similar to the multiset class, except in two regards:
  - The values in an unordered_multiset are not sorted
  - The unordered_multiset class has better performance
- You should use the unordered_multiset class instead of the multiset class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in ascending order
- The unordered_multiset class is declared in the <unordered_set> header file

STL Algorithms

- The STL provides a number of algorithms, implemented as function templates, in the `<algorithm>` header file.
- These functions perform various operations on ranges of elements.
- A range of elements is a sequence of elements denoted by two iterators:
  - The first iterator points to the first element in the range
  - The second iterator points to the end of the range (the element to which the second iterator points is not included in the range).

Categories of Algorithms in the STL

- Min/max algorithms
- Sorting algorithms
- Search algorithms
- Read-only sequence algorithms
- Copying and moving algorithms
- Swapping algorithms
- Replacement algorithms
- Removal algorithms
- Reversal algorithms
- Fill algorithms
- Rotation algorithms
- Shuffling algorithms
- Set algorithms
- Transformation algorithm
- Partition algorithms
- Merge algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm

Sorting

- The `sort` function:

  `sort(`*`iterator1`*`, `*`iterator2`*`);`

  `iterator1` and *`iterator2`* mark the beginning and end of a range of elements. The function sorts the range of elements in ascending order.

Searching

- The `binary_search` function:

```
binary_search(iterator1, iterator2, value);
```

iterator1 and *iterator2* mark the beginning and end of a range of elements that are sorted in ascending order. value is the value to search for. The function returns `true` if value is found in the range, or `false` otherwise.

Searching

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

int main()
{
    int searchValue;    // Value to search for

    // Create a vector of unsorted integers.
    vector<int> numbers = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};

    // Sort the vector.
    sort(numbers.begin(), numbers.end());

    // Display the vector.
    cout << "Here are the sorted values:\n";
    for (auto element : numbers)
        cout << element << " ";
    cout << endl;

    // Get the value to search for.
    cout << "Enter a value to search for: ";
    cin >> searchValue;

    // Search for the value.
    if (binary_search(numbers.begin(), numbers.end(), searchValue))
        cout << "That value is in the vector.\n";
    else
        cout << "That value is not in the vector.\n";

    return 0;
}
```

Detecting Permutations

- If a range has *N* elements, there are *N*! possible arrangements, or permutations, of those elements.
- For example, the range of integers 1, 2, 3 has six possible permutations:

```
1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1
```

Detecting Permutations

- The `is_permutation()` function determines whether one range of elements is a permutation of another range of elements.

```
is_permutation(iterator1, iterator2, iterator3)
```

  - *`iterator1`* and *`iterator2`* mark the beginning and end of the first range of elements.
  - *`iterator3`* marks the beginning of the second range of elements, assumed to have the same number of elements as the first range.
  - The function returns `true` if the second range is a permutation of the first range, or `false` otherwise.

Plugging Your Own Functions into an Algorithm

- Many of the function templates in the STL are designed to accept function pointers as arguments.
- This allows you to "plug" one of your own functions into the algorithm.
- For example:

```
for_each(iterator1, iterator2, function)
```

  - *iterator1* and *iterator2* mark the beginning and end of a range of elements.
  - *function* is the name of a function that accepts an element as its argument.
  - The `for_each()` function iterates over the range of elements, passing each element as an argument to *function*.

Plugging Your Own Functions into an Algorithm

- For example, consider this function:

```
void doubleNumber(int &n)
{
    n = n * 2;
}
```

- And this code snippet:

```
vector<int> numbers = { 1, 2, 3, 4, 5 };

// Display the numbers before doubling.
for (auto element : numbers)
   cout << element << " ";
cout << endl;

// Double the value of each vector element.
for_each(numbers.begin(), numbers.end(), doubleNumber);

// Display the numbers before doubling.
for (auto element : numbers)
   cout << element << " ";
cout << endl;
```

This passes each element of the `numbers` vector to the `doubleNumber` function.

Plugging Your Own Functions into an Algorithm

- Another example:

```
count_if(iterator1, iterator2, function)
```

  - *iterator1* and *iterator2* mark the beginning and end of a range of elements.
  - *function* is the name of a function that accepts an element as its argument, and returns either `true` or `false`.
  - The `count_if()` function iterates over the range of elements, passing each element as an argument to *function*.
  - The `count_if` function returns the number of elements for which function returns `true`.

Plugging Your Own Functions into an Algorithm

```
 1 #include <iostream>
 2 #include <vector>
 3 #include <algorithm>
 4 using namespace std;
 5
 6 // Function prototypes
 7 bool isNegative(int);
 8
 9 int main()
10 {
11    // Create a vector of ints.
12    vector<int> numbers = { 0, 99, 120, -33, 10, 8, -1, 101 };
13
14    // Get the number of elements that are negative.
15    int negatives = count_if(numbers.begin(), numbers.end(), isNegative);
16
17    // Display the results.
18    cout << "There are " << negatives << " negative elements.\n";
19    return 0;
20 }
21
22 // isNegative function
23 bool isNegative(int n)
24 {
25    bool status = false;
26
27    if (n < 0)
28       status = true;
29
30    return status;
31 }
```

Algorithms for Set Operations

- The STL provides function templates for basic mathematical set operations.

STL Function Template	Description
`set_union`	Finds the union of two sets, which is a set that contains all the elements of both sets, excluding duplicates.
`set_intersection`	Finds the intersection of two sets, which is a set that contains only the elements that are found in both sets.
`set_difference`	Finds the difference of two sets, which is the set of elements that appear in one set, but not the other.
`set_symmetric_difference`	Finds the symmetric difference of two sets, which is the set of elements that appear in one set, but not both.
`set_includes`	Determines whether one set includes another.

Function Objects

- A function object is an object that acts like a function.
  - It can be called
  - It can accept arguments
  - It can return a value
- Function objects are also known as *functors*

Function Objects

- To create a function object, you write a class that overloads the ( ) operator.

```
1   #ifndef SUM_H
2   #define SUM_H
3
4   class Sum
5   {
6   public:
7       int operator()(int a, int b)
8       { return a + b; }
9   };
10  #endif
```

Accepts two int arguments

Returns an int

Function Objects

```
#include <iostream>
#include "Sum.h"
using namespace std;

int main()
{
    // Local variables
    int x = 10;
    int y = 2;
    int z = 0;

    // Create a Sum object.
    Sum sum;

    // Call the sum function object.
    z = sum(x, y);

    // Display the result.
    cout << z << endl;

    return 0;
}
```

Anonymous Function Objects

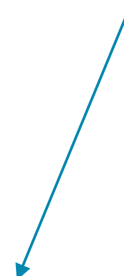
- Function objects can be called at the point of their creation, without being given a name. Consider this class:

```
1   #ifndef IS_EVEN_H
2   #define IS_EVEN_H
3
4   class IsEven
5   {
6   public:
7       bool operator()(int x)
8       { return x % 2 == 0; }
9   };
10  #endif
```

Anonymous Function Objects

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>
4   #include "IsEven.h"
5   using namespace std;
6
7   int main()
8   {
9       // Create a vector of ints.
10      vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };
11
12      // Get the number of elements that even.
13      int evenNums = count_if(v.begin(), v.end(), IsEven());
14
15      // Display the results.
16      cout << "The vector contains " << evenNums << " even numbers.\n";
17      return 0;
18  }
```

An `IsEven` object is created here, but not given a name. It is anonymous.



Predicate Terminology

- A function or function object that returns a Boolean value is called a *predicate*.
- A predicate that takes only one argument is called a *unary predicate*.
- A predicate that takes two arguments is called a *binary predicate*.
- This terminology is used in much of the available C++ documentation and literature.

Lambda Expressions

- A lambda expression is a compact way of creating a function object without having to write a class declaration.
- It is an expression that contains only the logic of the object's `operator()` member function.
- When the compiler encounters a lambda expression, it automatically generates a function object in memory, using the code that you provide in the lambda expression for the `operator()` member function.

Lambda Expressions

- General format:

  `[](`*`parameter list`*`) { `*`function body`*` }`

- The `[ ]` is known as the lambda introducer. It marks the beginning of a lambda expression.
- *`parameter list`* is a list of parameter declarations for the function object's `operator()` member function.
- *`function body`* is the code that should be the body of the object's `operator()` member function.

Lambda Expressions

- Example: a lambda expression for a function object that computes the sum of two integers:
```
[](int a, int b) { return x + y; }
```
- Example: a lambda expression for a function object that determines whether an integer is even is:
```
[](int x) { return x % 2 == 0; }
```
- Example: a lambda expression for a function object that takes an integer as input and prints the square of that integer:
```
[](int a) { cout << a * a << " "; }
```

Lambda Expressions

- When you call a lambda expression, you write a list of arguments, enclosed in parentheses, right after the expression.
- For example, the following code snippet displays 7, which is the sum of the variables x and y:

```
int x = 2;
int y = 5;
cout << [](int a, int b) {return a + b;}(x, y) << endl;
```

Lambda Expressions

- The following code segment counts the even numbers in a `vector`:

```
// Create a vector of ints.
vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };
// Get the number of elements that are even.
int evenNums = count_if(v.begin(), v.end(), [](int x) {return x % 2 == 0;});
// Display the results.
cout << "The vector contains " << evenNums << " even numbers.\n";
```

- Because lambda expressions generate function objects, you can assign a lambda expression to a variable and then call it through the variable's name:

```
auto sum = [](int a, int b) {return a + b;};
int x = 2;
int y = 5;
int z = sum(x, y);
```

Lambda Expressions

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

int main()
{
    // Create a vector of ints.
    vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };

    // Use a lambda expression to create a function object.
    auto isEven = [](int x) { return x % 2 == 0; };

    // Get the number of elements that even.
    int evenNums = count_if(v.begin(), v.end(), isEven);

    // Display the results.
    cout << "The vector contains " << evenNums << " even numbers.\n";
    return 0;
}
```

Functional Classes in the STL

- The STL library defines a number of classes that you can instantiate to create function objects in your program.
- To use these classes, you must `#include` the `<functional>` header file.
- Table 17-15 lists a few of the functional classes:

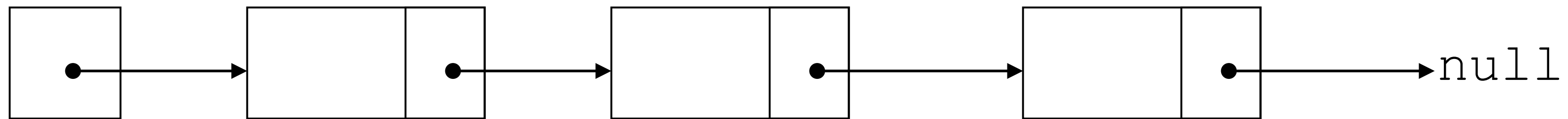
Table 17-15 STL Function Object Classes

Functional Class	Description
`less<T>`	`less<T>()(T a, T b)` is true if and only if `a < b`
`less_equal<T>`	`less_equal()(T a, T b)` is true if an only if `a <= b`
`greater<T>`	`greater<T>()(T a, T b)` is true if and only if `a > b`
`greater_equal<T>`	`greater_equal<T>()(T a, T b)` is true if and only if `a >= b`

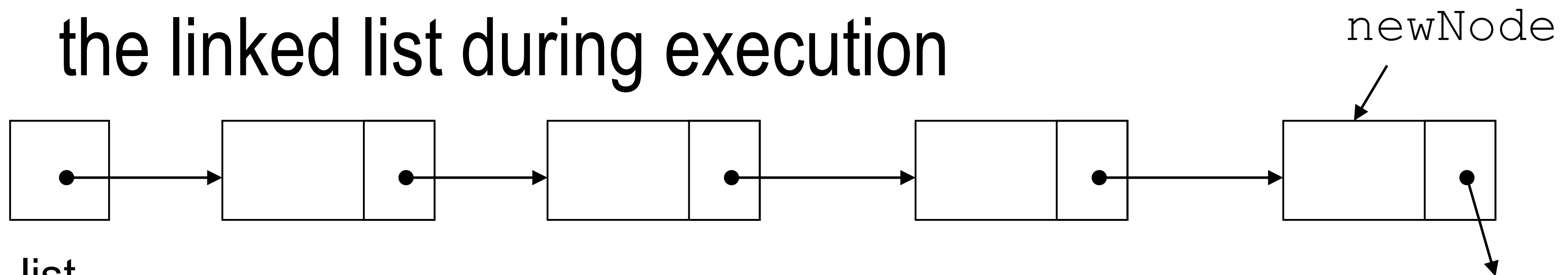
5. Data Structures

Linked List ADT

- Linked list: set of data structures (nodes) that contain references to other data structures

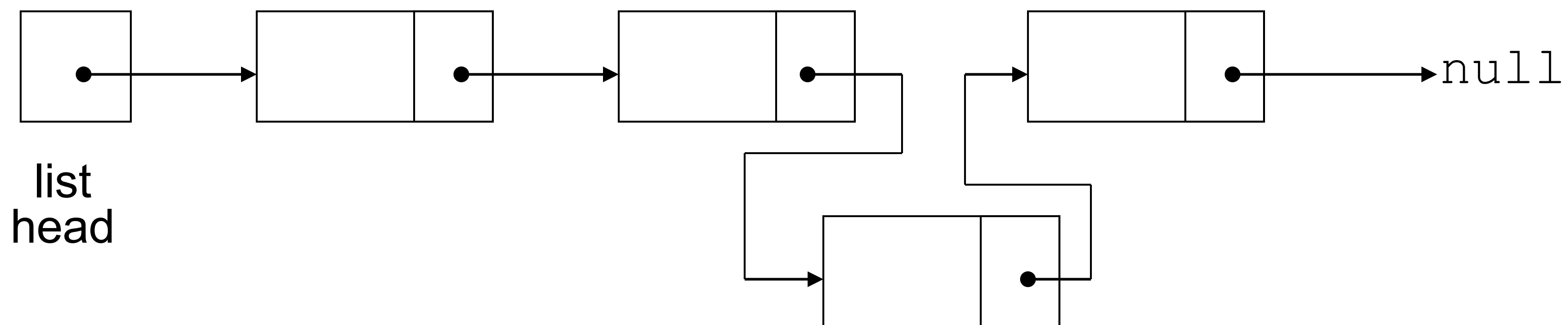


- References may be addresses or array indices
- Data structures can be added to or removed from the linked list during execution



Linked Lists vs. Arrays and Vectors

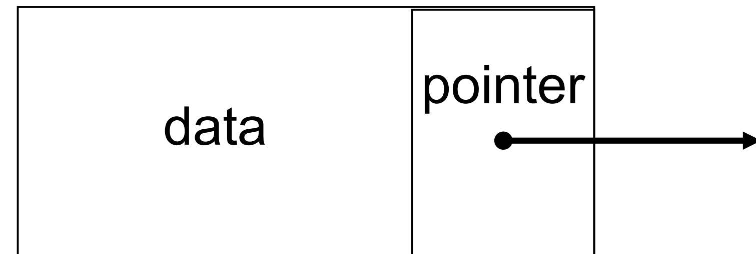
- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily



Node Organization - Declaring a Node

- A node contains:
  - data: one or more data fields – may be organized as structure, object, etc.
  - a pointer that can point to another node
- Declare a node:

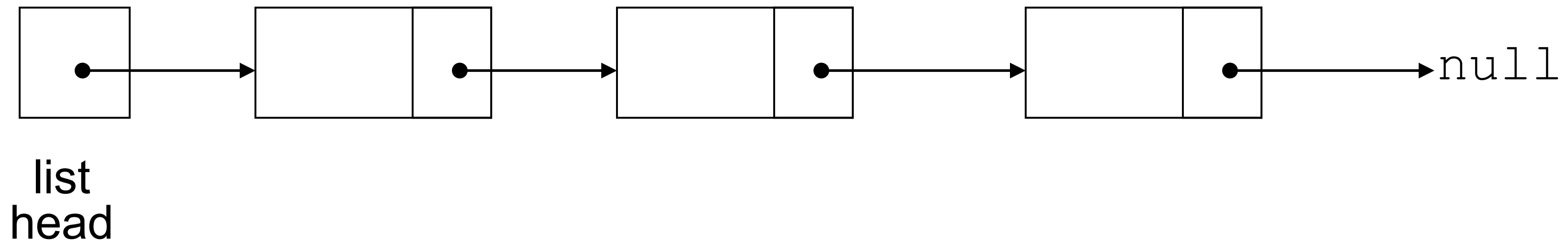
```
struct ListNode {
        int data;
        ListNode *next;
};
```



- No memory is allocated at this time

Linked List Organization

- Linked list contains 0 or more nodes:

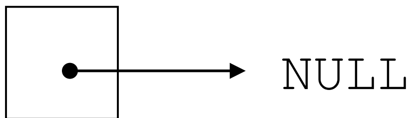


- Has a list head to point to first node
- Last node points to `null (address 0)`

Empty List

- If a list currently contains 0 nodes, it is the <u>empty list</u>
- In this case the list head points to `null`

list
head

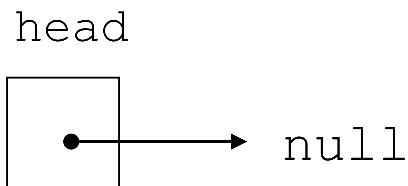


Defining a Linked List

- Define a pointer for the head of the list:

```
ListNode *head = nullptr;
```

- Head pointer initialized to `nullptr` to indicate an empty list



The Null Pointer

- Is used to indicate end-of-list
- Should always be tested for before using a pointer:

```
ListNode *p;
while (!p)
```

Linked List Operations

- Basic operations:
  - append a node to the end of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list

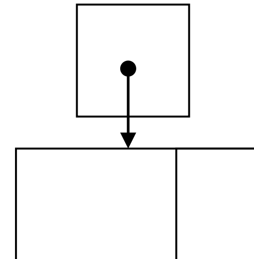
Linked List Example - NumberList.h

```
 1  // Specification file for the NumberList class
 2  #ifndef NUMBERLIST_H
 3  #define NUMBERLIST_H
 4
 5  class NumberList
 6  {
 7  private:
 8     // Declare a structure for the list
 9     struct ListNode
10     {
11        double value;           // The value in this node
12        struct ListNode *next;  // To point to the next node
13     };
14
15     ListNode *head;            // List head pointer
16
17  public:
18     // Constructor
19     NumberList()
20        { head = nullptr; }
21
22     // Destructor
23     ~NumberList();
24
25     // Linked list operations
26     void appendNode(double);
27     void insertNode(double);
28     void deleteNode(double);
29     void displayList() const;
30  };
31  #endif
```

Create a New Node

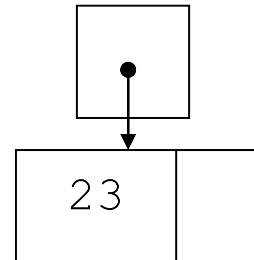
- Allocate memory for the new node:

```
newNode = new ListNode;
```



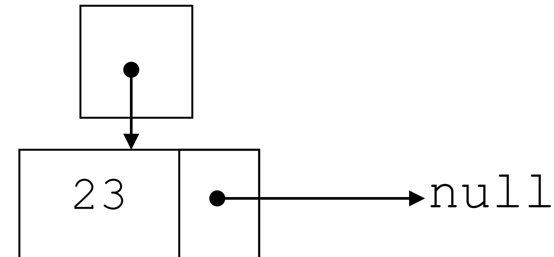
- Initialize the contents of the node:

```
newNode->value = num;
```



- Set the pointer field to `nullptr`:

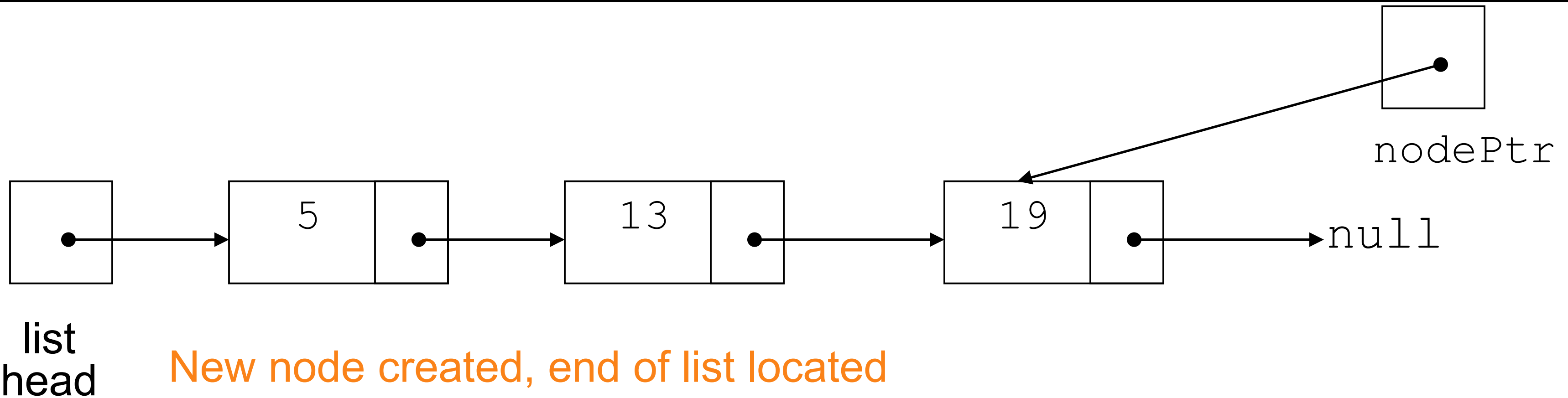
```
newNode->next = nullptr;
```



Appending a Node

- Add a node to the end of the list
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node



23

null

newNode

nodePtr

5

13

19

list

head

New node added to end of list

23

null

newNode

C++ code for Appending a Node

```
11  void NumberList::appendNode(double num)
12  {
13     ListNode *newNode;   // To point to a new node
14     ListNode *nodePtr;   // To move through the list
15
16     // Allocate a new node and store num there.
17     newNode = new ListNode;
18     newNode->value = num;
19     newNode->next = nullptr;
20
21     // If there are no nodes in the list
22     // make newNode the first node.
23     if (!head)
```

C++ code for Appending a Node (Continued)

```
24          head = newNode;
25       else  // Otherwise, insert newNode at end.
26       {
27          // Initialize nodePtr to head of list.
28          nodePtr = head;
29
30          // Find the last node in the list.
31          while (nodePtr->next)
32             nodePtr = nodePtr->next;
33
34          // Insert newNode as the last node.
35          nodePtr->next = newNode;
36       }
37    }
```

Appending a Node

```
// This program demonstrates a simple append
// operation on a linked list.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    // Define a NumberList object.
    NumberList list;

    // Append some values to the list.
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    return 0;
}
```

Inserting a Node into a Linked List

- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

Inserting a Node into a Linked List

New node created, correct position located

Inserting a Node into a Linked List

New node inserted in order in the linked list

```
69  void NumberList::insertNode(double num)
70  {
71      ListNode *newNode;                   // A new node
72      ListNode *nodePtr;                   // To traverse the list
73      ListNode *previousNode = nullptr; // The previous node
74
75      // Allocate a new node and store num there.
76      newNode = new ListNode;
77      newNode->value = num;
78
79      // If there are no nodes in the list
80      // make newNode the first node
81      if (!head)
82      {
83          head = newNode;
84          newNode->next = nullptr;
85      }
86      else // Otherwise, insert newNode
87      {
88          // Position nodePtr at the head of list.
89          nodePtr = head;
90
```

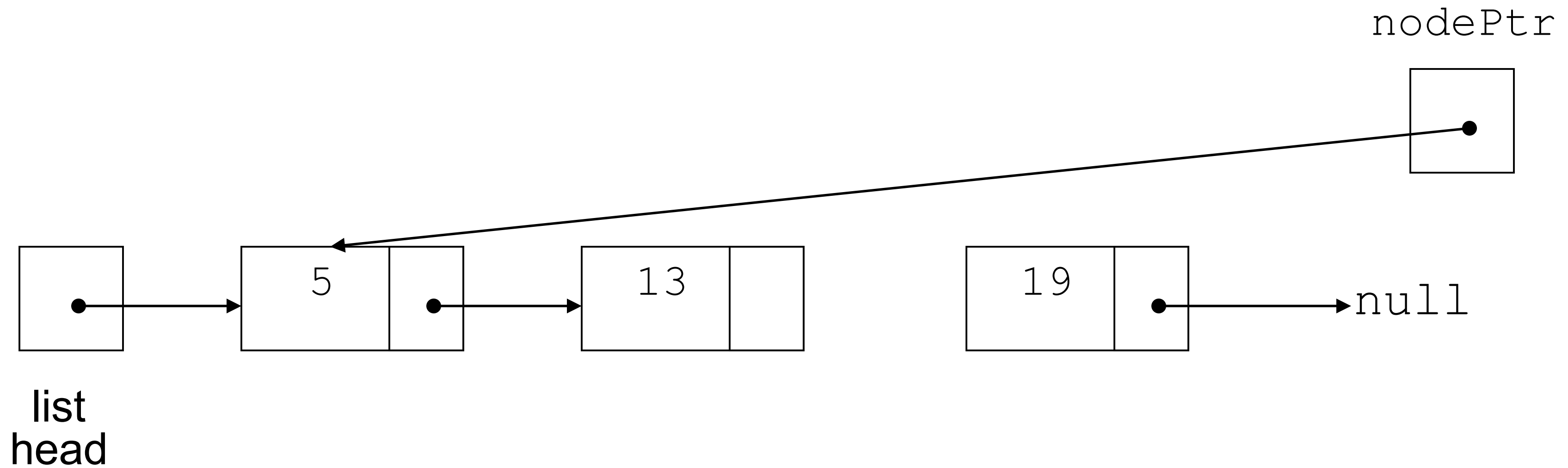
```
 91            // Initialize previousNode to nullptr.
 92            previousNode = nullptr;
 93
 94            // Skip all nodes whose value is less than num.
 95            while (nodePtr != nullptr && nodePtr->value < num)
 96            {
 97                previousNode = nodePtr;
 98                nodePtr = nodePtr->next;
 99            }
100
101            // If the new node is to be the 1st in the list,
102            // insert it before all other nodes.
103            if (previousNode == nullptr)
104            {
105                head = newNode;
106                newNode->next = nodePtr;
107            }
108            else // Otherwise insert after the previous node.
109            {
110                previousNode->next = newNode;
111                newNode->next = nodePtr;
112            }
113        }
114    }
```

```
// This program demonstrates the insertNode member function.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    // Define a NumberList object.
    NumberList list;

    // Build the list with some values.
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);

    // Insert a node in the middle of the list.
    list.insertNode(10.5);

    // Display the list
    list.displayList();
    return 0;
}
```

Traversing a Linked List

- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not a null pointer
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while

Traversing a Linked List

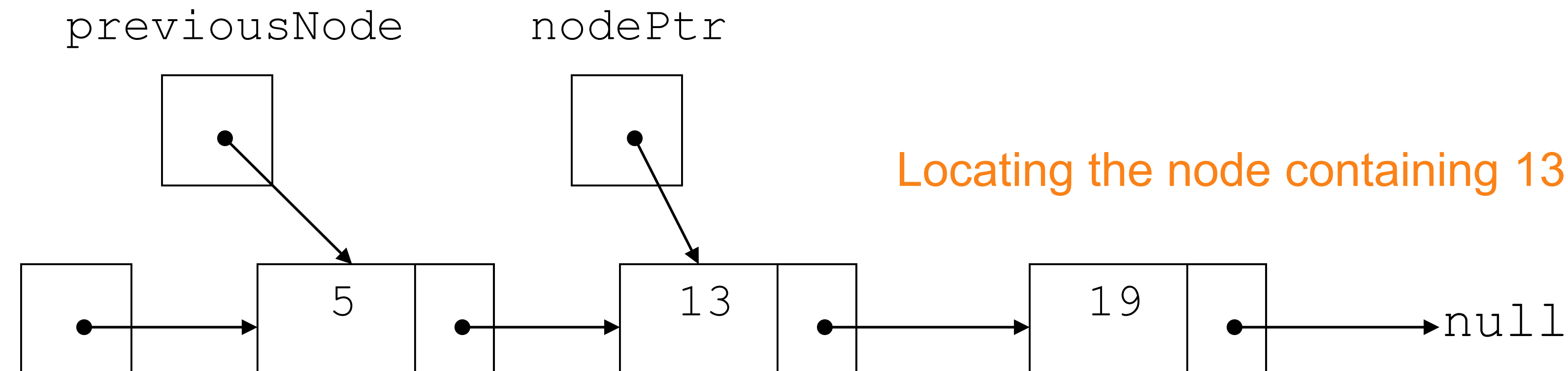


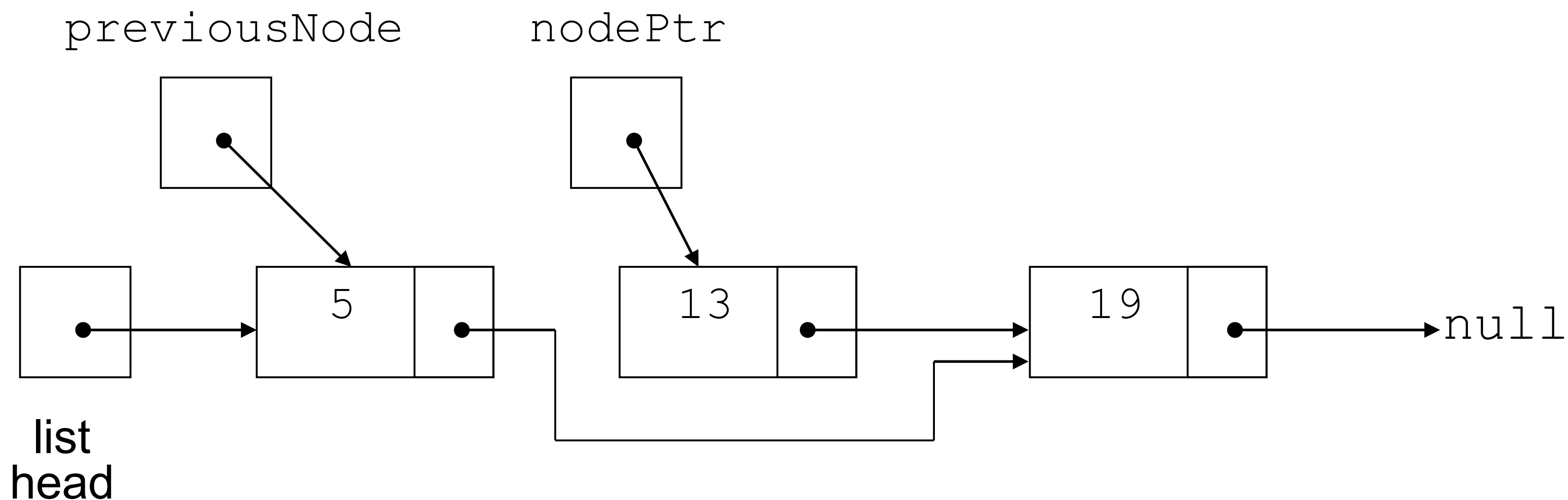
`nodePtr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to the null pointer, and the list traversal stops

Deleting a Node

- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

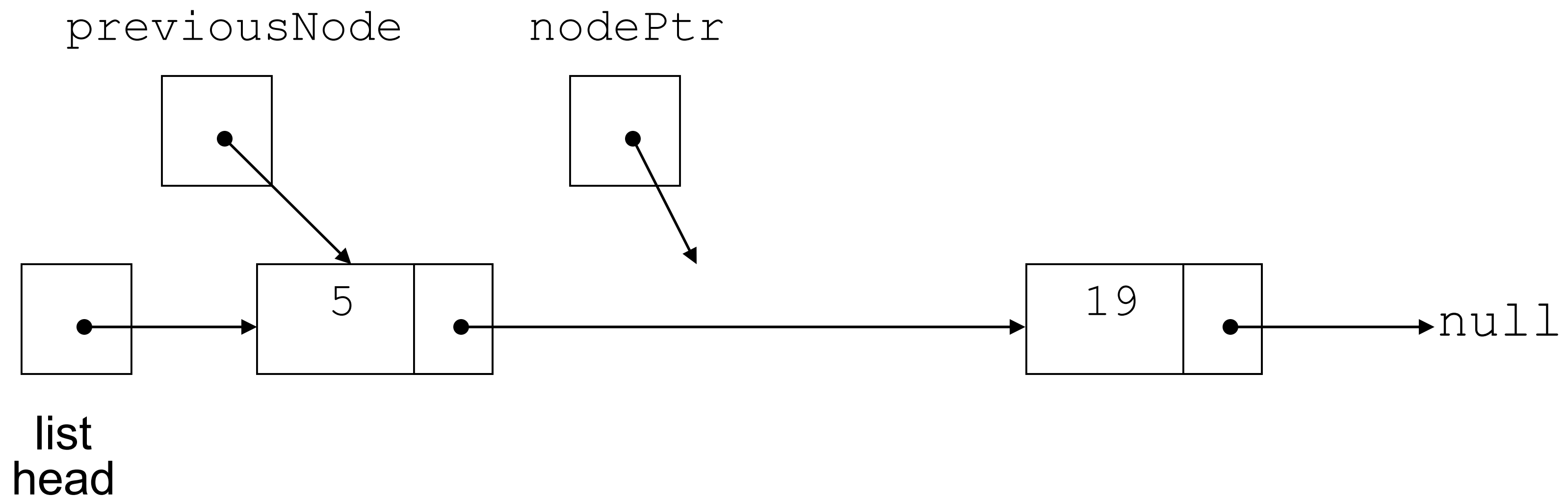
Deleting a Node





Adjusting pointer around the node to be deleted

Deleting a Node



Linked list after deleting the node containing 13

Deleting a Node

```
void NumberList::deleteNode(double num)
{
    ListNode *nodePtr;       // To traverse the list
    ListNode *previousNode;  // To point to the previous node

    // If the list is empty, do nothing.
    if (!head)
        return;
    // Determine if the first node is the one.
    if (head->value == num)
    {
        nodePtr = head->next;
        delete head;
        head = nodePtr;
    }
    else
    {
        // Initialize nodePtr to head of list
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr != nullptr && nodePtr->value != num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }
        // If nodePtr is not at the end of the list,
        // link the previous node to the node after
        // nodePtr, then delete nodePtr.
        if (nodePtr)
        {
            previousNode->next = nodePtr->next;
            delete nodePtr;
        }
    }
}
```

Deleting a Node

```
// This program demonstrates the deleteNode member function.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    // Define a NumberList object.
    NumberList list;

    // Build the list with some values.
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    // Display the list.
    cout << "Here are the initial values:\n";
    list.displayList();
    cout << endl;

    // Delete the middle node.
    cout << "Now deleting the node in the middle.\n";
    list.deleteNode(7.9);

    // Display the list.
    cout << "Here are the nodes left.\n";
    list.displayList();
    cout << endl;

    // Delete the last node.
    cout << "Now deleting the last node.\n";
    list.deleteNode(12.6);

    // Display the list.
    cout << "Here are the nodes left.\n";
    list.displayList();
    cout << endl;

    // Delete the only node left in the list.
    cout << "Now deleting the only remaining node.\n";
    list.deleteNode(2.5);

    // Display the list.
    cout << "Here are the nodes left.\n";
    list.displayList();
    return 0;
}
```

Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node's memory
- Set the list head to `nullptr`

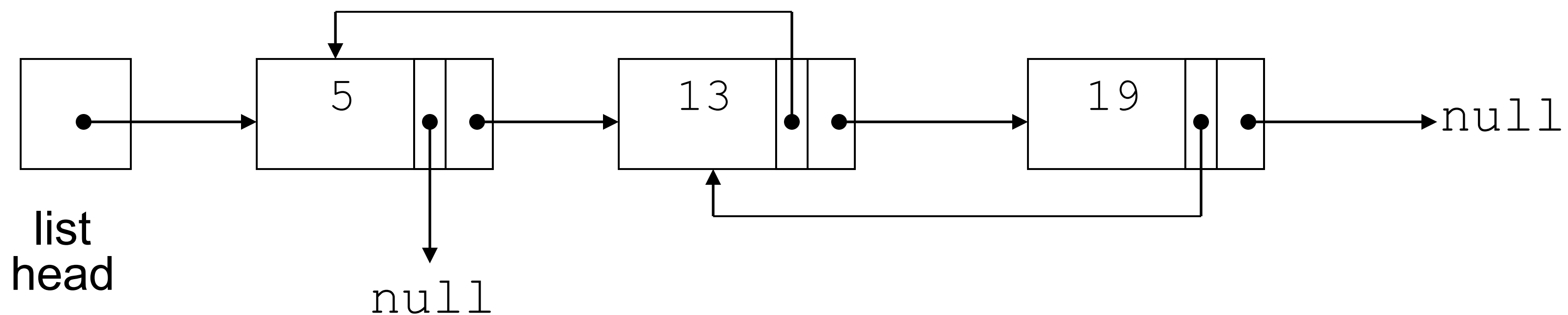
```
NumberList::~NumberList()
{
    ListNode *nodePtr;    // To traverse the list
    ListNode *nextNode;   // To point to the next node

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr is not at the end of the list...
    while (nodePtr != nullptr)
    {
        // Save a pointer to the next node.
        nextNode = nodePtr->next;

        // Delete the current node.
        delete nodePtr;

        // Position nodePtr at the next node.
        nodePtr = nextNode;
    }
}
```

A Linked List Template

- When declaring a linked list, must specify the type of data to be held in each node
- Using templates, can declare a linked list that can hold data type determined at list definition time

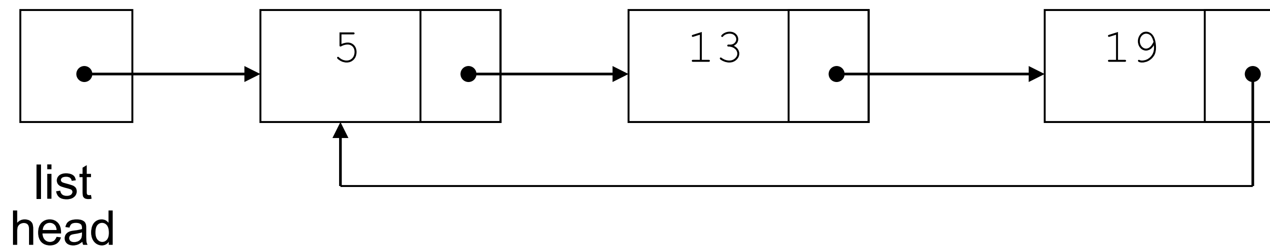
Variations of the Linked List

- Other linked list organizations:
  - doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



Variations of the Linked List

- Other linked list organizations:
  - circular linked list: the last node in the list points back to the first node in the list, not to the null pointer



The STL `list` Container

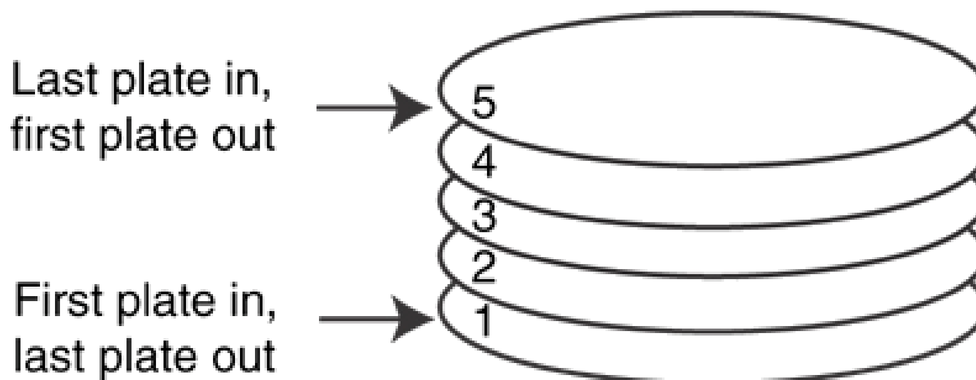
- Template for a doubly linked list
- Member functions for
  - locating beginning, end of list: `front, back, end`
  - adding elements to the list: `insert, merge, push_back, push_front`
  - removing elements from the list: `erase, pop_back, pop_front, unique`
- See this link for a list of constructors and member functions

The STL `forward_list` Container

- Template for a singly linked list
- You can only step forward in a `forward_list`.
- A `forward_list` uses slightly less memory than a `list`, and has takes slightly less time for inserting and removing nodes.
- Provides most, but not all, of the same member functions as the `list` container

Introduction to the Stack ADT

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria
  - return addresses for function calls
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list



Stack Operations and Functions

- Operations:
  - push: add a value onto the top of the stack
  - pop: remove a value from the top of the stack
- Functions:
  - `isFull`: `true` if the stack is currently full, *i.e.*, has no more space to hold additional elements
  - isEmpty: `true` if the stack currently contains no elements

Dynamic Stacks

- Grow and shrink as necessary
- Can't ever be full as long as memory is available
- Implemented as a linked list

Implementing a Stack

- Programmers can program their own routines to implement stack functions
- See `DynIntStack` class for an example.
- Can also use the implementation of stack available in the STL

The STL `stack` container

- Stack template can be implemented as a `vector`, a linked list, or a `deque`
- Implements `push`, `pop`, and `empty` member functions
- Implements other member functions:
  - `size`: number of elements on the stack
  - `top`: reference to element on top of the stack

Defining a `stack`

- Defining a stack of `char`s, named `cstack`, implemented using a `vector`:

```
stack< char, vector<char>> cstack;
```

- implemented using a list:

```
stack< char, list<char>> cstack;
```

- implemented using a `deque`:

```
stack< char > cstack;
```

- When using a compiler that is older than C++ 11, be sure to put spaces between the angled brackets that appear next to each other.

```
stack< char, vector<char> > cstack;
```

Introduction to the Queue ADT

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

Queue Locations and Operations

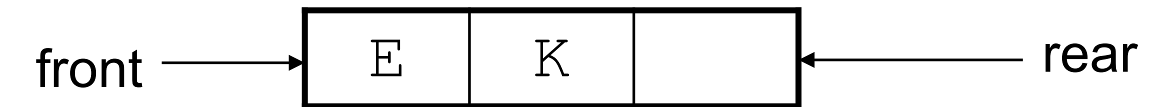
- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

Queue Operations - Example

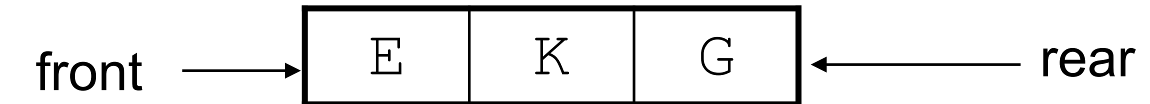
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

- `enqueue('K');`

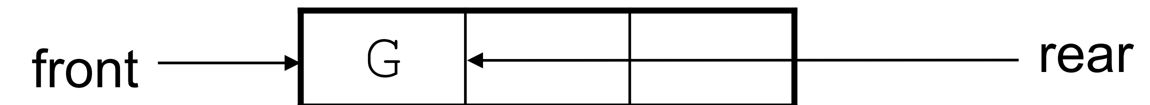


- `enqueue('G');`



`// remove E`
- `dequeue();`

`// remove K`
- `dequeue();`

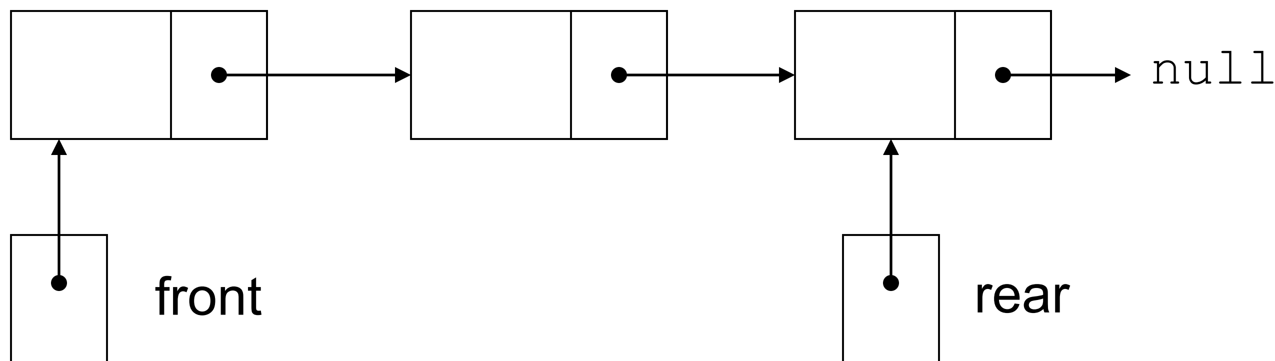


`dequeue` Issue, Solutions

- When removing an element from a queue, remaining elements must shift to front
- Solutions:
  - Let front index move as elements are removed (works as long as rear index is not at end of array)
  - Use above solution, and also let rear index "wrap around" to front of array, treating array as circular instead of linear (more complex enqueue, dequeue code)

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



Implementing a Queue

- Programmers can program their own routines to implement queue operations
- See the `DynIntQue` class in the book for an example of a dynamic queue
- Can also use the implementation of queue and dequeue available in the STL

The STL `deque` and `queue` Containers

- `deque`: a double-ended queue. Has member functions to enqueue (`push_back`) and dequeue (`pop_front`)
- `queue`: container ADT that can be used to provide queue as a `vector`, list, or `deque`. Has member functions to enque (`push`) and dequeue (`pop`)

Defining a `queue`

- Defining a queue of `char`s, named cQueue, implemented using a deque:

```
deque<char> cQueue;
```

- implemented using a queue:

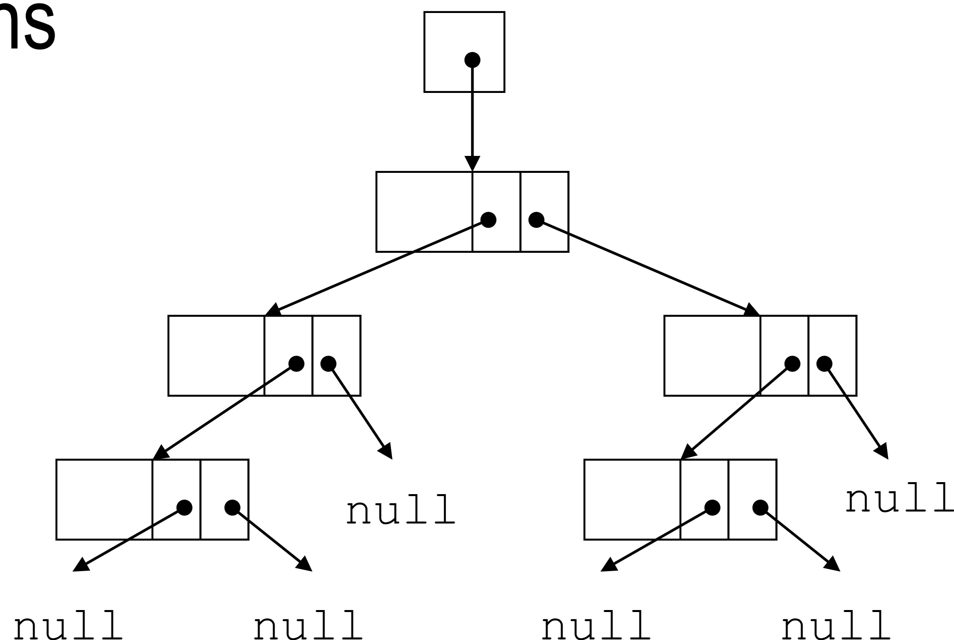
```
queue<char> cQueue;
```

- implemented using a `list`:

```
queue<char, list<char>> cQueue;
```

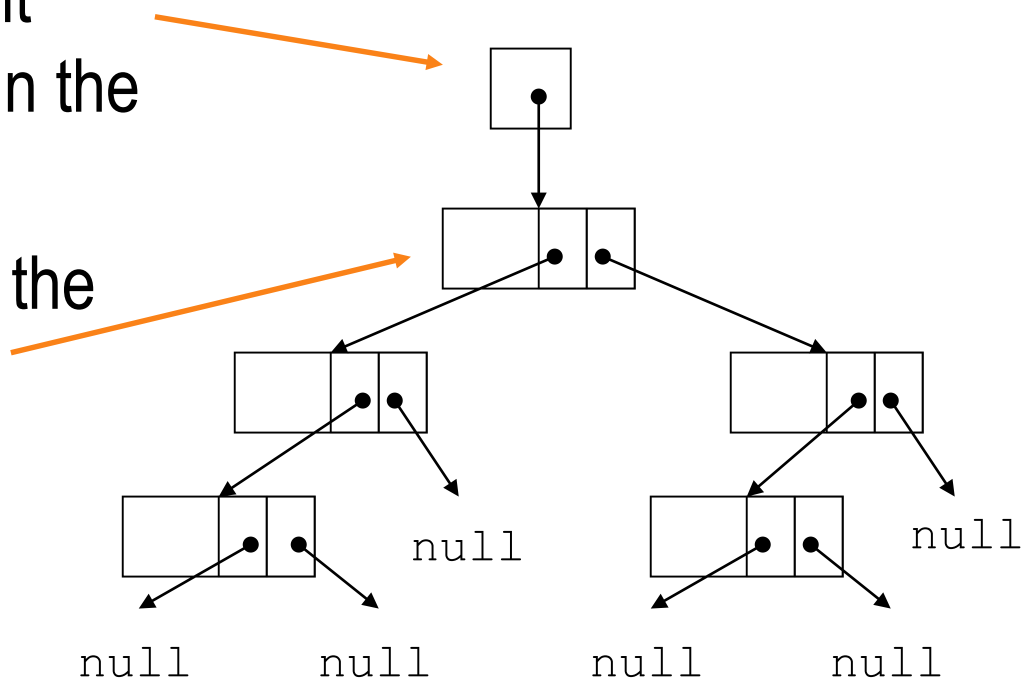
Definition and Application of Binary Trees

- Binary tree: a nonlinear linked list in which each node may point to 0, 1, or two other nodes
- Each node contains one or more data fields and two pointers



Binary Tree Terminology

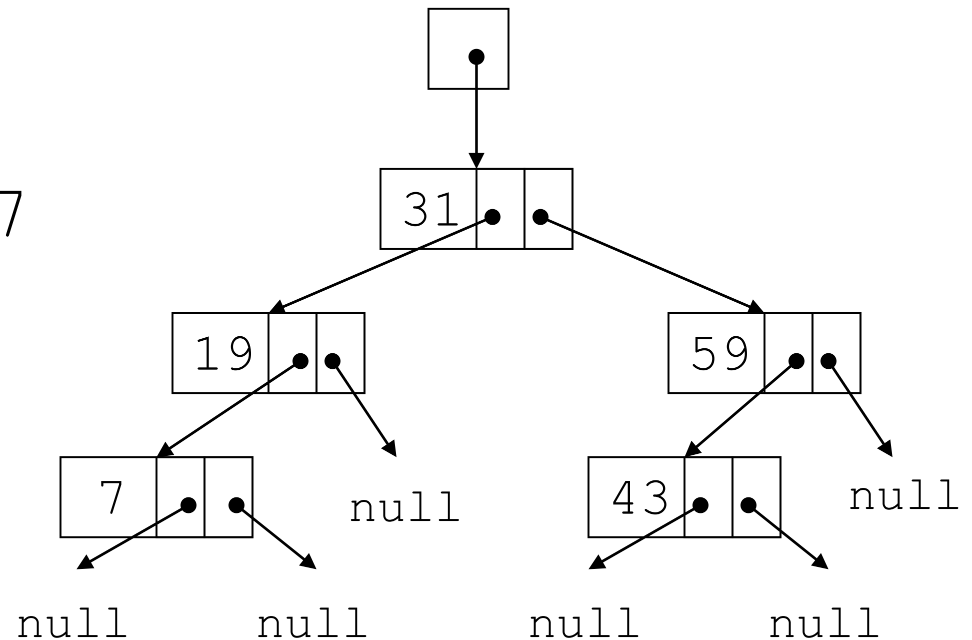
- Tree pointer: like a head pointer for a linked list, it points to the first node in the binary tree
- Root node: the node at the top of the tree



Binary Tree Terminology

- Leaf nodes: nodes that have no children

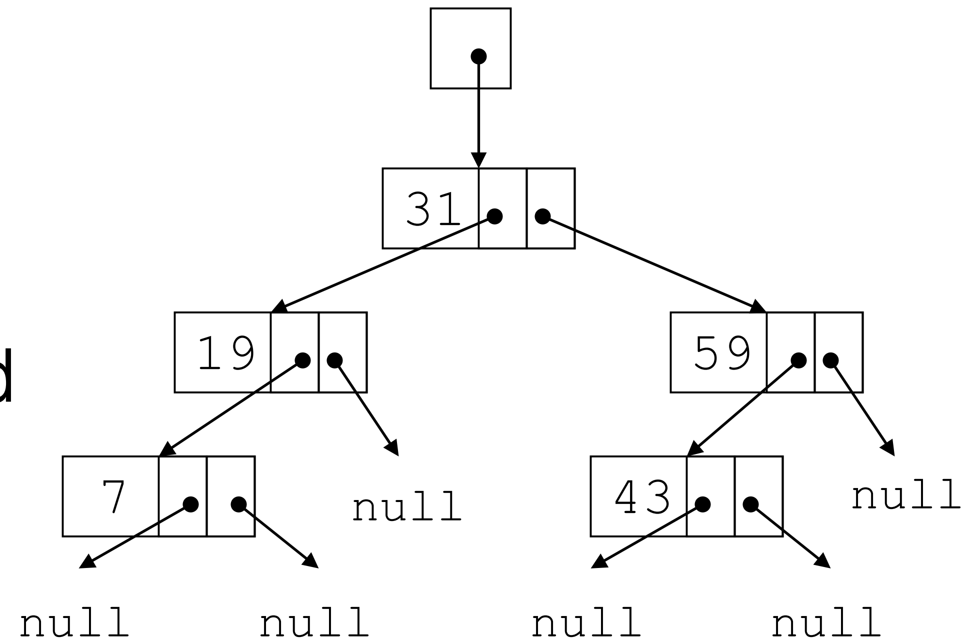
  The nodes containing `7` and `43` are leaf nodes



Binary Tree Terminology

- Child nodes, children: nodes below a given node

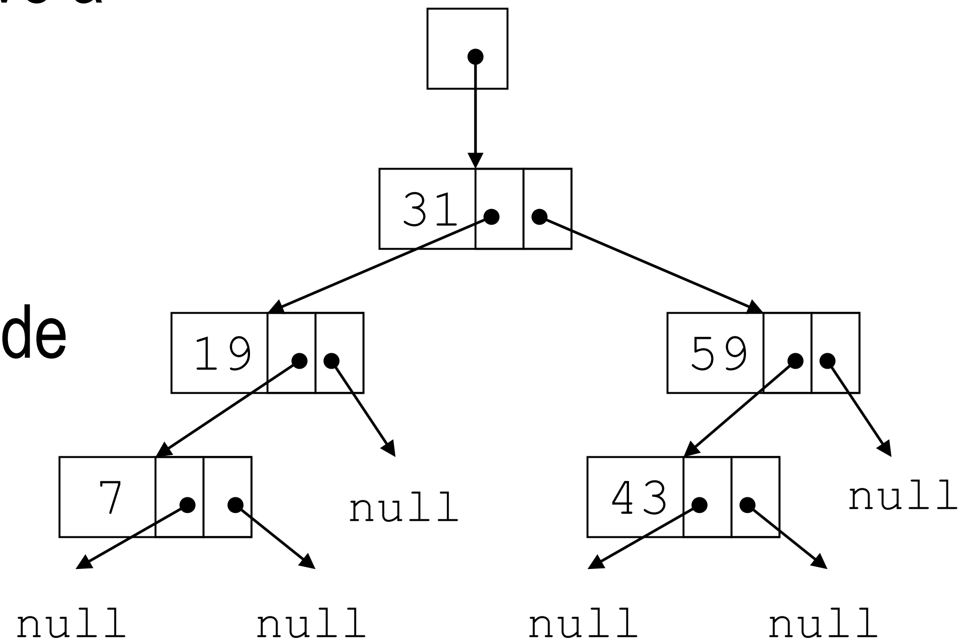
  The children of the node containing `31` are the nodes containing `19` and `59`



Binary Tree Terminology

- Parent node: node above a given node

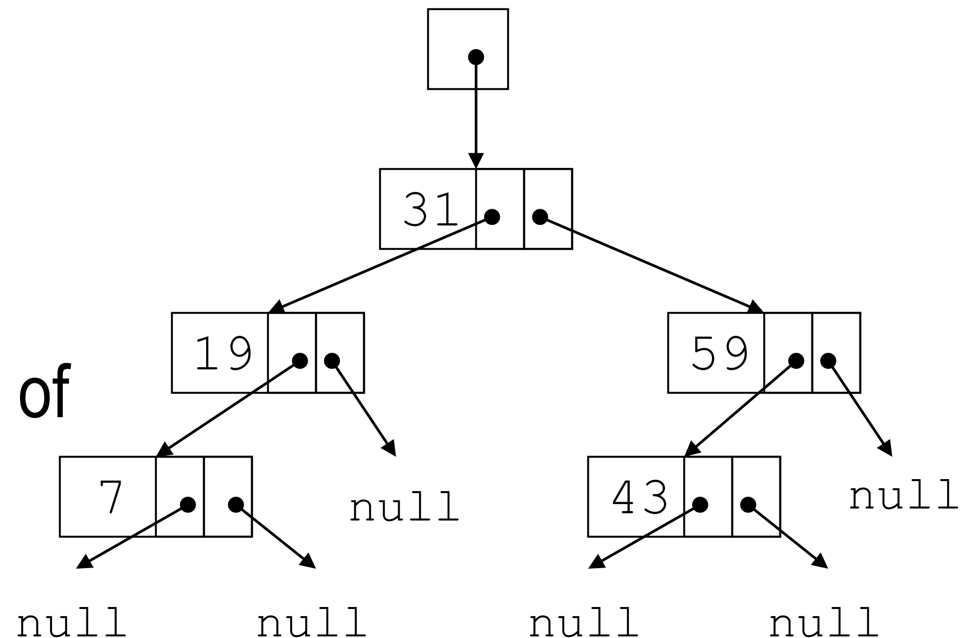
  The parent of the node containing `43` is the node containing `59`



Binary Tree Terminology

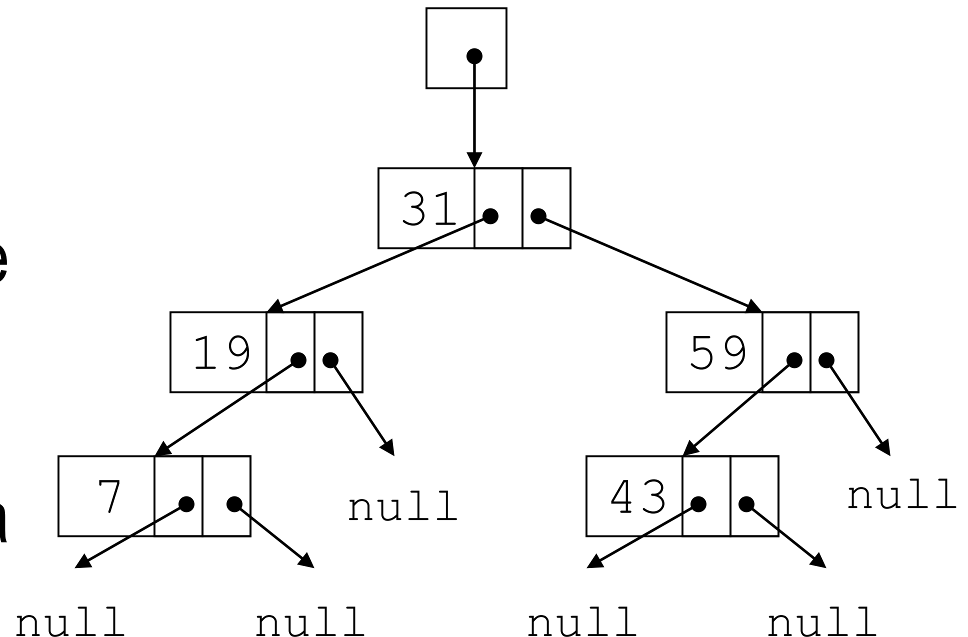
- Subtree: the portion of a tree from a node down to the leaves

  The nodes containing `19` and `7` are the left subtree of the node containing `31`



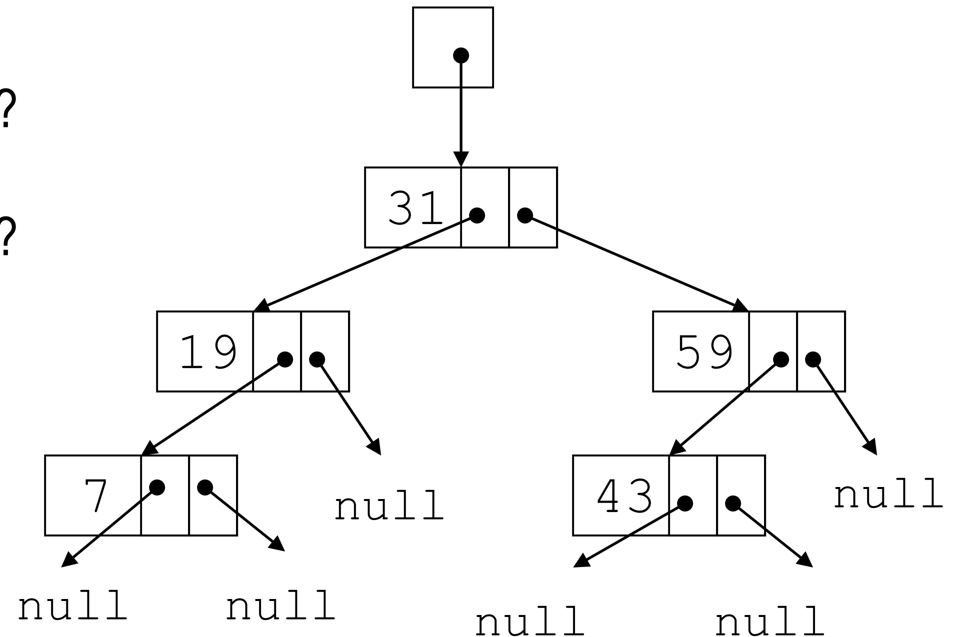
Uses of Binary Trees

- Binary search tree: data organized in a binary tree to simplify searches
- Left subtree of a node contains data values < the data in the node
- Right subtree of a node contains values > the data in the node



Searching in a Binary Tree

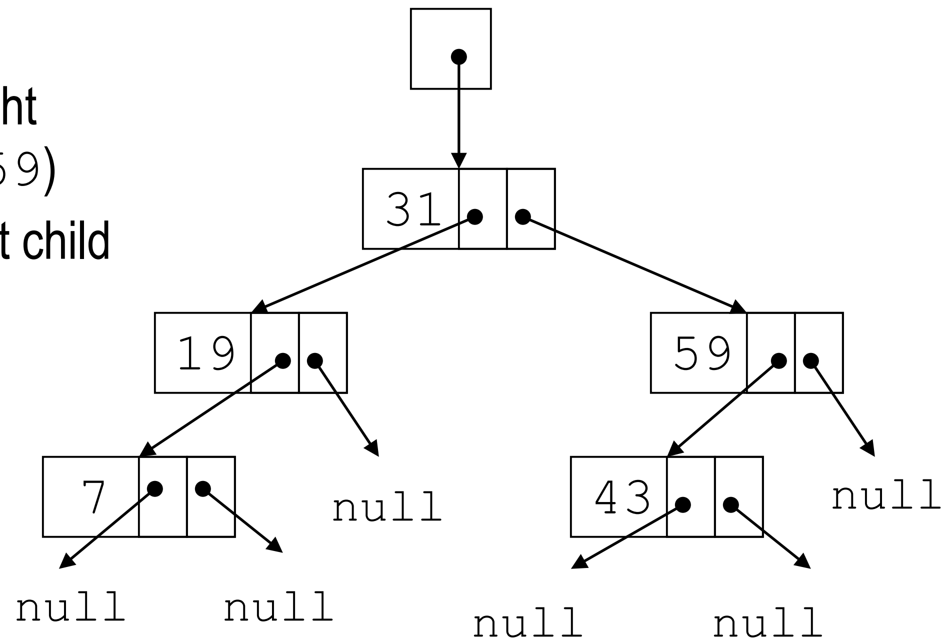
1) Start at root node
2) Examine node data:
   a) Is it desired value? Done
   b) Else, is desired data < node data? Repeat step 2 with left subtree
   c) Else, is desired data > node data? Repeat step 2 with right subtree
3) Continue until desired value found or a null pointer reached



Searching in a Binary Tree

To locate the node containing `43`,

- Examine the root node (`31`) first
- Since `43 > 31`, examine the right child of the node containing `31`, (`59`)
- Since `43 < 59`, examine the left child of the node containing `59`, (`43`)
- The node containing 43 has been found



Binary Search Tree Operations

- Create a binary search tree – organize data into a binary search tree
- Insert a node into a binary tree – put node into tree in its correct position to maintain order
- Find a node in a binary tree – locate a node with particular data value
- Delete a node from a binary tree – remove a node and adjust links to maintain binary tree

Binary Search Tree Node

- A node in a binary tree is like a node in a linked list, with two node pointer fields:

```
struct TreeNode
{
   int value;
   TreeNode *left;
   TreeNode *right;
}
```

Creating a New Node

- Allocate memory for new node:

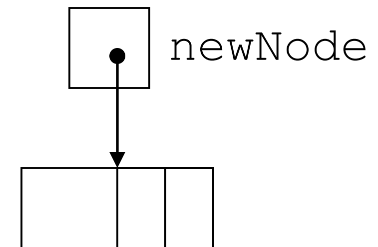
```
newNode = new TreeNode;
```

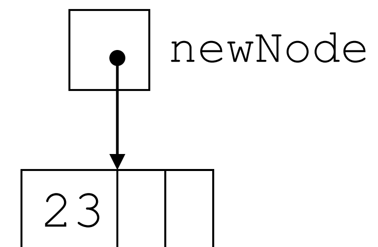
- Initialize the contents of the node:

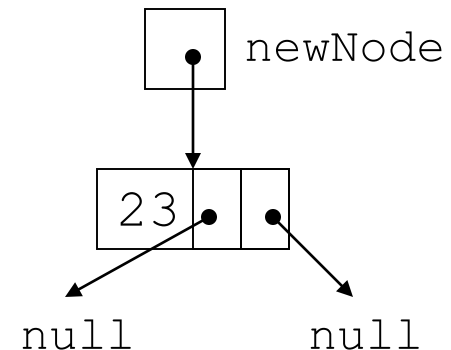
```
newNode->value = num;
```

- Set the pointers to `nullptr`:

```
newNode->Left
= newNode->Right
= nullptr;
```







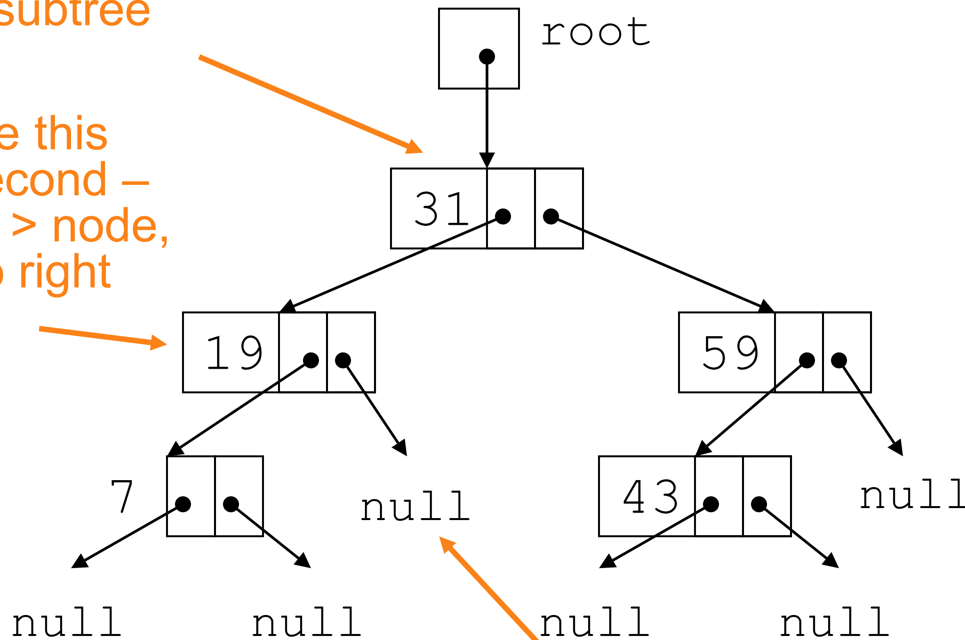
Inserting a Node in a Binary Search Tree

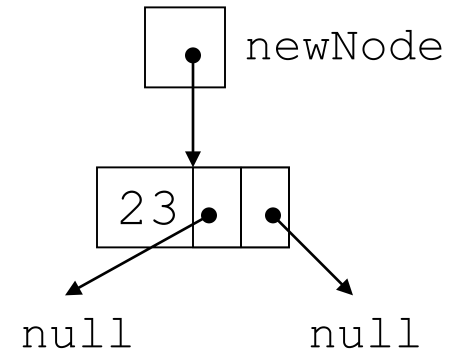
1) If tree is empty, insert the new node as the root node
2) Else, compare new node against left or right child, depending on whether data value of new node is < or > root node
3) Continue comparing and choosing left or right subtree unitl null pointer found
4) Set this null pointer to point to new node

Inserting a Node in a Binary Search Tree

Examine this node first – value is < node, so go to left subtree

Examine this node second – value is > node, so go to right subtree





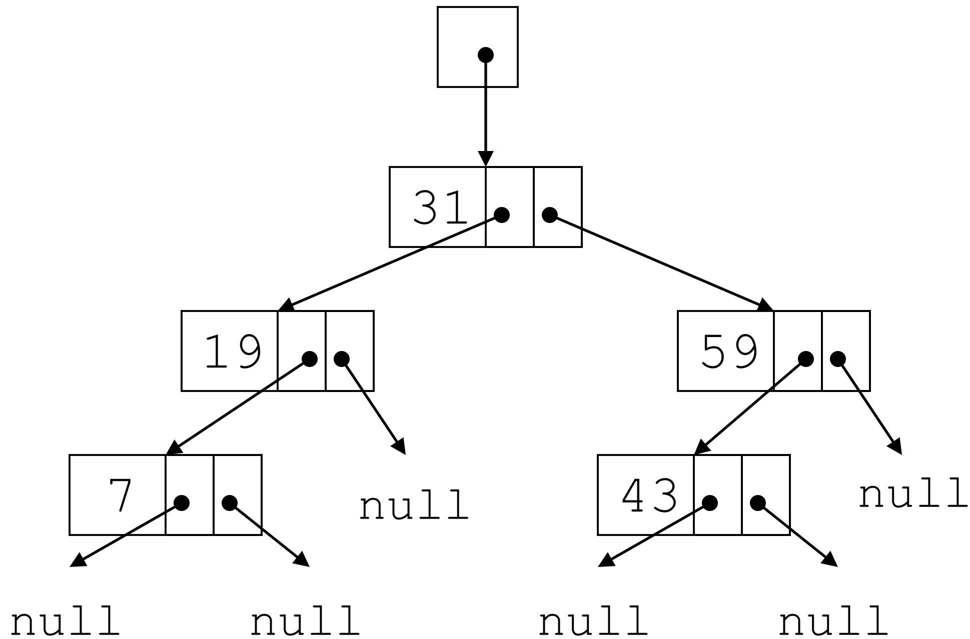
Since the right subtree is `null`, insert here

Traversing a Binary Tree

Three traversal methods:

1) Inorder:
   a) Traverse left subtree of node
   b) Process data in node
   c) Traverse right subtree of node
2) Preorder:
   a) Process data in node
   b) Traverse left subtree of node
   c) Traverse right subtree of node
3) Postorder:
   a) Traverse left subtree of node
   b) Traverse right subtree of node
   c) Process data in node

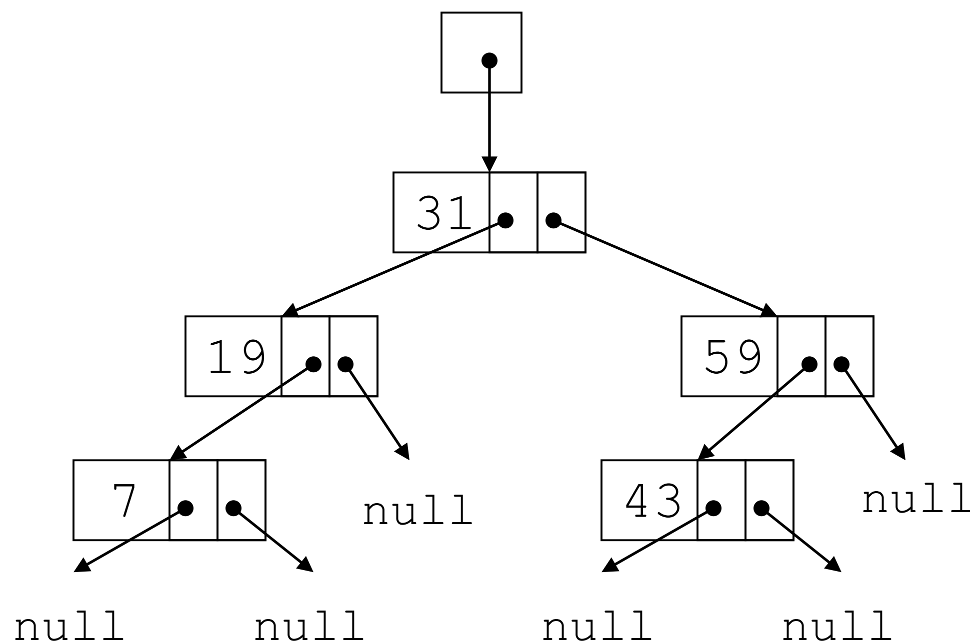
Traversing a Binary Tree



TRAVERSAL METHOD	NODES VISITED IN ORDER
Inorder	7, 19, 31, 43, 59
Preorder	31, 19, 7, 59, 43
Postorder	7, 19, 43, 59, 31

Searching in a Binary Tree

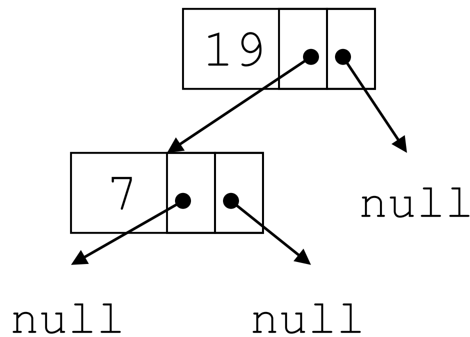
- Start at root node, traverse the tree looking for value
- Stop when value found or null pointer detected
- Can be implemented as a `bool` function



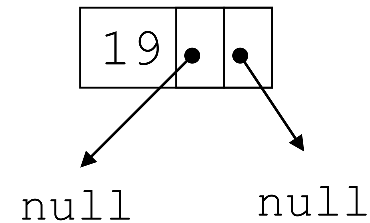
Search for `43`? return `true`
Search for `17`? return `false`

Deleting a Node from a Binary Tree – Leaf Node

- If node to be deleted is a leaf node, replace parent node's pointer to it with the null pointer, then delete the node



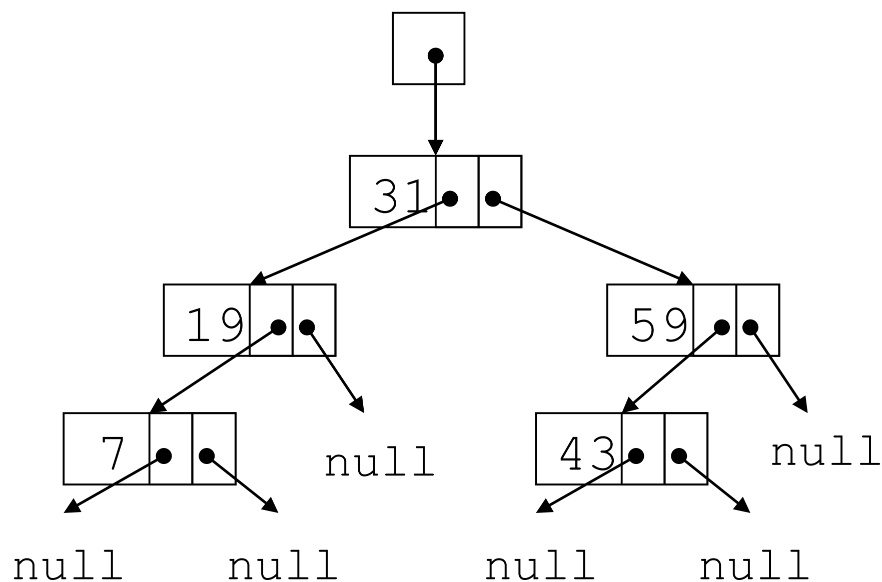
Deleting node with 7 – before deletion



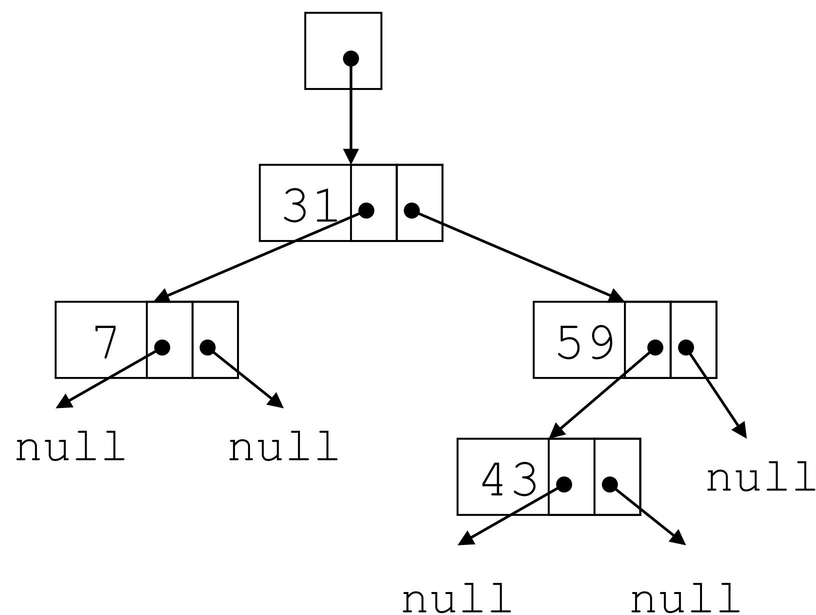
Deleting node with 7 – after deletion

Deleting a Node from a Binary Tree – One Child

- If node to be deleted has one child node, adjust pointers so that parent of node to be deleted points to child of node to be deleted, then delete the node



Deleting node with 19 – before deletion

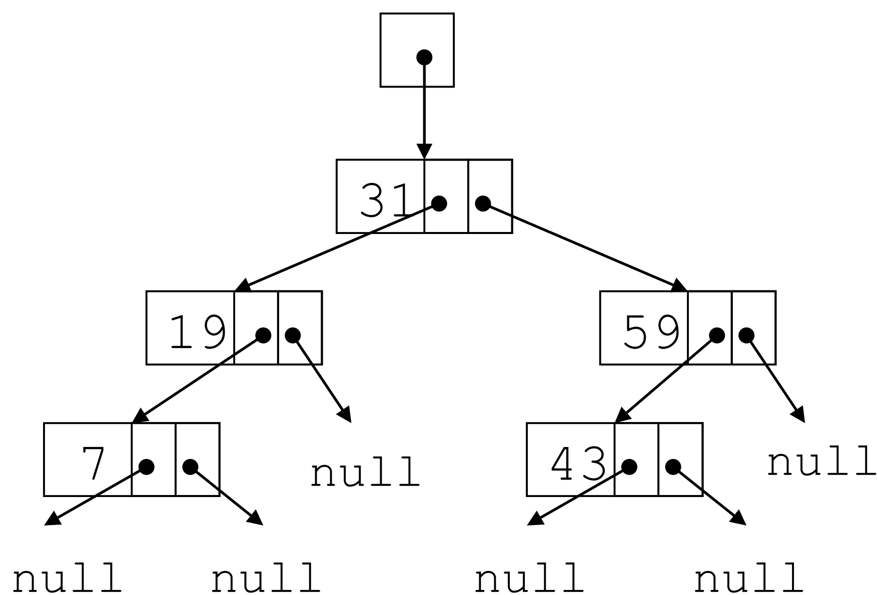


Deleting node with 19 – after deletion

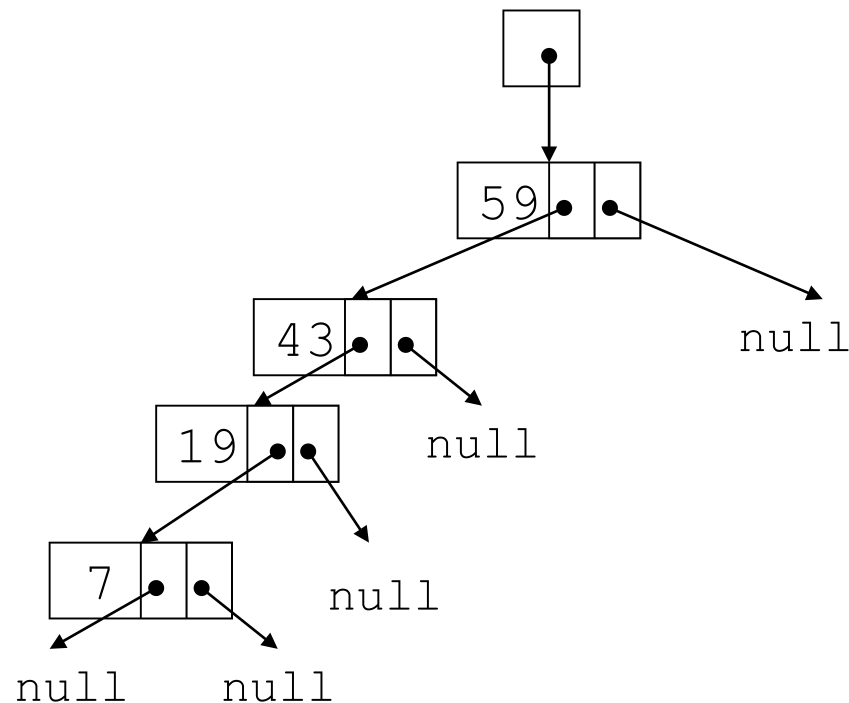
Deleting a Node from a Binary Tree – Two Children

- If node to be deleted has left and right children,
  - ‘Promote’ one child to take the place of the deleted node
  - Locate correct position for other child in subtree of promoted child
- Convention in text: promote the right child, position left subtree underneath

Deleting a Node from a Binary Tree – Two Children



Deleting node with 31 – before deletion



Deleting node with 31 – after deletion

Template Considerations for Binary Search Trees

- Binary tree can be implemented as a template, allowing flexibility in determining type of data stored
- Implementation must support relational operators >, <, and == to allow comparison of nodes

Referencias

- Starting out with C++ : from control structures through objects, Tony Gaddis, Pearson
- C++ Programming Tutorial
- A Tour of C++