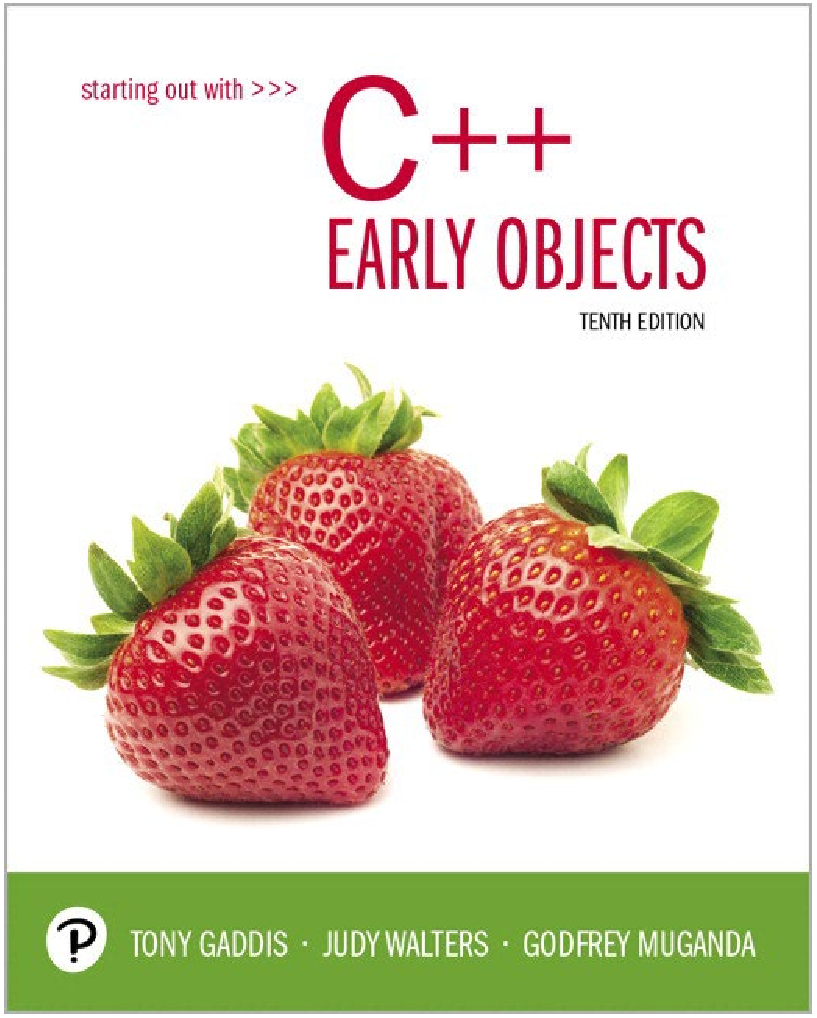
Starting Out with C++ Early Objects

Tenth Edition



Chapter 7

Introduction to Classes and Objects

Topics 1 of 2

Topics 2 of 2

7.1 Abstract Data Types

- Are programmer-created data types that specify
  - the legal values that can be stored
  - the operations that can be done on the values
- The user of an abstract data type (ADT) does not need to know any implementation details (*e.g.*, how the data is stored or how the operations on it are carried out)

Abstraction in Software Development

- Abstraction allows a programmer to design a solution to a problem and to use data items without concern for how the data items are implemented
- This has already been encountered in the book:
  - To use the `pow` function, you need to know what inputs it expects and what kind of results it produces
  - You do not need to know how it works

Abstraction and Data Types

- Abstraction: a definition that captures general characteristics without details

  ex: An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral

- Data Type: defines the kind of values that can be stored and the operations that can be performed on the values

7.2 Object-Oriented Programming

- Procedural programming uses variables to store data, and focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.
- Object-oriented programming is based on objects that encapsulate the data and the functions that operate with and on the data.

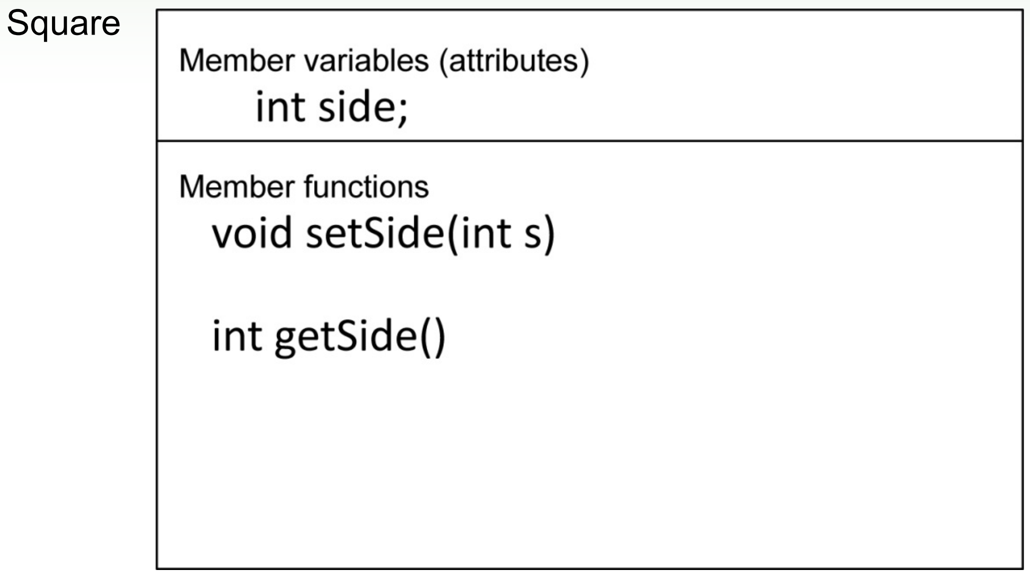
Object-Oriented Programming Terminology 1 of 2

- **object**: software entity that combines data and functions that act on the data in a single unit
- **attributes**: the data items of an object, stored in **member variables**
- **member functions (methods)**: procedures/ functions that act on the attributes of the class

Object-Oriented Programming Terminology 2 of 2

- **data hiding**: restricting data access to certain members of an object. The intent is to allow only member functions to directly access and modify the object's data
- **encapsulation**: the bundling of an object's data and procedures into a single entity

Object Example



Square object's data item: **`side`**

Square object's functions: **`setSide`** - set the size of the side of the square, **`getSide`** - return the size of the side of the square

Why Hide Data?

- Protection – Member functions provide a layer of protection against inadvertent or deliberate data corruption
- Need-to-know – A programmer can use the data via the provided member functions. As long as the member functions return correct information, the programmer needn't worry about implementation details.

7.3 Introduction to Classes 1 of 2

- **Class**: a programmer-defined data type used to define objects
- It is a pattern for creating objects

ex:

```
string fName, lName;
```

This creates two objects of the `string` class

Introduction to Classes 2 of 2

- Class declaration format:

```
class className
{
    declaration;

};
```

Note the required ;

Access Specifiers

- Used to control access to members of the class.
- Each member is declared to be either

  `public:` can be accessed by functions outside of the class

  or

  `private:` can only be called by or accessed by functions that are members of the class

Class Example

Access specifiers

```
class Square
{
  private:
     int side;
  public:
     void setSide(int s)
     { side = s; }
     int getSide()
     { return side; }
};
```

More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is **`private`**

7.4 Creating and Using Objects

- An object is an instance of a class
- It is defined just like other variables

```
Square sq1, sq2;
```

- It can access members using dot operator

```
sq1.setSide(5);
cout << sq1.getSide();
```

Types of Member Functions

- **Acessor, get, getter function**: uses but does not modify a member variable

  ex: `getSide`

- **Mutator, set, setter function**: modifies a member variable

  ex: `setSide`

7.5 Defining Member Functions

- Member functions are part of a class declaration
- You can place entire function definition inside the class declaration,

  or
- You can place just the prototype inside the class declaration and write the function definition after the class

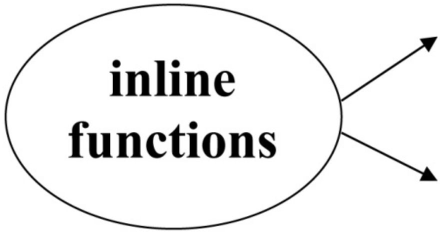
Defining Member Functions Inside the Class Declaration

- Member functions defined inside the class declaration are called inline functions
- Only very short functions, like the one below, should be inline functions

```
int getSide()
{ return side; }
```

Inline Member Function Example

```
class Square
{
  private:
    int side;
  public:
    void setSide(int s)
    { side = s; }
    int getSide()
    { return side; }
};
```



Defining Member Functions After the Class Declaration

- Use a function prototype in the class declaration
- In the function definition, precede the function name with the class name and scope resolution operator (::)

```
int Square::getSide()
 {
    return side;
 }
```

Conventions and a Suggestion

Conventions:

- Member variables are usually **`private`**
- Accessor and mutator functions are usually **`public`**
- Use ‘get’ in the name of accessor functions, ‘set’ in the name of mutator functions

Suggestion: calculate values to be returned in accessor functions when possible, to minimize the potential for stale data

Tradeoffs of Inline vs. Regular Member Functions

- When a regular function is called, control passes to the called function
  - the compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into the program in place of the call when the program is compiled
  - This makes a larger executable program, but
  - There is less function call overhead, and possibly faster execution

7.6 Constructors

- A **constructor** is a member function that is automatically called when an object of the class is created
- Is can be used to initialize data members
- It must be a `public` member function
- It must be named the same as the class
- It must have no return type

Constructor – 2 Examples

Inline:

```
class Square
{
  . . .
  public:
    Square(int s)
    { side = s; }
  . . .
};
```

Declaration outside the class:

```
Square(int);   //prototype
               //in class
Square::Square(int s)
{
   side = s;
}
```

Overloading Constructors

- A class can have more than 1 constructor
- Overloaded constructors in a class must have different parameter lists

```
Square();

Square(int);
```

The Default Constructor

- Constructors can have any number of parameters, including none
- A **default constructor** is one that takes no arguments either due to
  - No parameters or
  - All parameters have default values
- If a class has any programmer-defined constructors, it should have a programmer -defined default constructor

Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square()           // default
    { side = 1; }      // constructor

    // Other member
    // functions go here
};
```

Has no parameters

Another Default Constructor Example

```
class Square
{
  private:
     int side;

  public:
     Square(int s = 1) // default
     { side = s; }     // constructor

     // Other member
     // functions go here
};
```

Has a parameter but it has a default value

Invoking a Constructor

- To create an object using the default constructor, use no argument list and no `()`

```
Square square1;
```

- To create an object using a constructor that has parameters, include an argument list

```
Square square2(8);
```

Member Initialization List 1 of 2

- Member variables can be initialized using a member initialization list: a comma-separated list of member variables and initial values following the header of the constructor and before the body.
- Examples:

```
Square() : side(1)
{  }

Square(int size) : side(size)
{  }
```

Member Initialization List 2 of 2

Notes:

- `:` is required between header and initialization list
- no `;` at the end of the initialization list
- initialization list precedes the function body. It can be on the same or the following line of the header.
- The initialization list may accomplish all that is needed in a constructor. In this case, the body of the constructor is empty. The { } are still required.

In-Place Member Initialization List

- Starting with C++ 11, member variables can be initialized at the time that they are defined in the class declaration. This provides default values those variables for all objects created from the class.
- Default values can be overridden by using a constructor that takes parameters for the variables that have been initialized in-place.

Constructor Delegation

- Starting with C++ 11, a constructor in a class can call another constructor in the same class. This is called constructor delegation.
- The notation is similar to member initialization list
- Example:

```
// default constructor
 Square(): Square(1)
 {  }
 Square(int s)
 { side = s; }
```

Constructor Delegation Notes

- The default constructor delegates the work to the constructor that takes one argument.
- It calls the non-default constructor and passes the initial value as an argument.

```
// default constructor
   Square(): Square(1)
    { }
    Square(int s)
    { side = s; }
```

7.7 Destructors

- Is a public member function automatically called when an object is destroyed
- The destructor name is ~*className*, *e.g.*,

  ```
  ~Square
  ```

- It has no return type
- It takes no arguments
- Only 1 destructor is allowed per class
  (*i.e.*, it cannot be overloaded)

7. 8 Private Member Functions

- A **`private`** member function can only be called by another member function of the same class
- It is used for internal processing by the class, not for use outside of the class

7.9 Passing Objects to Functions

- A class object can be passed as an argument to a function.
- When it is passed by value, the function makes a local copy of the object. The original object in the calling environment is unaffected by actions in the function.
- When passed by reference, the function can use ‘set’ functions to modify the object in the calling environment.

Notes on Passing Objects 1 of 2

- Using a value parameter for an object can slow down a program and waste space

- Using a reference parameter speeds up the program. However, it allows the function to modify the data in the parameter, which is the same as the argument in the calling part of the program. This may not be desirable.

Notes on Passing Objects 2 of 2

- To save space and time while protecting parameter data that should not be changed, use a `const` reference parameter in the header and the prototype:

```
  void showData(const Square &s)
                              // header
```

- In order to for the showData function to call `Square` member functions, those functions must use `const` in their prototype and header:

```
 int Square::getSide() const;
```

Returning an Object from a Function

- A function can return an object

```
Square initSquare();    // prototype
Square s1 = initSquare();// call
```

- The function must create an object
  - for internal use
  - to use with the **`return`** statement

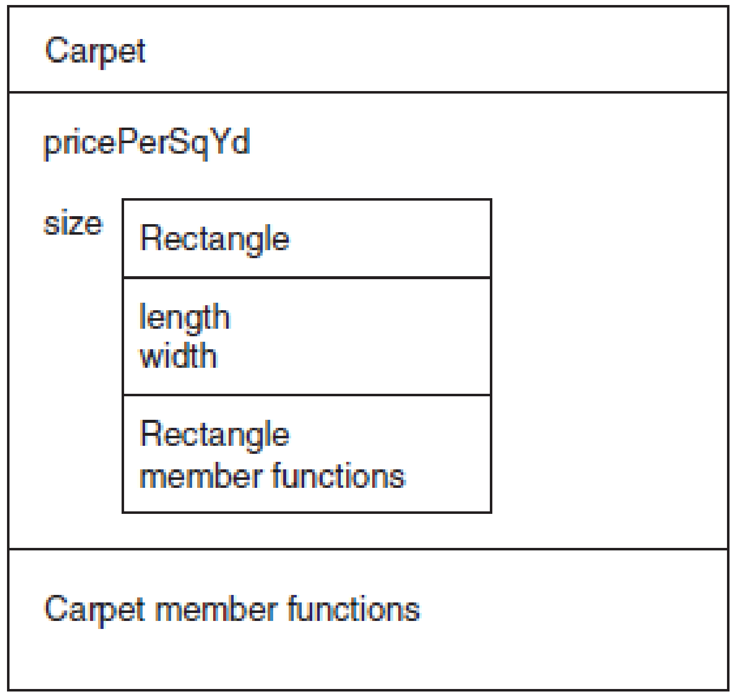
Returning an Object Example

```
Square initSquare()
{
   Square s;     // local object
   int inputSize;
   cout << "Enter the length of side: ";
   cin >> inputSize;
   s.setSide(inputSize);
   return s;
}
```

7.10 Object Composition 1 of 2

- This occurs when an object is a member variable of another object.
- It is often used to design complex objects whose members are simpler objects
- ex. (from book): Define a rectangle class. Then, define a carpet class and use a rectangle object as a member of a carpet object.

Object Composition 2 of 2



7.11 Separating Class Specification, Implementation, and Client Code

Separating the class declaration, member function definitions, and the program that uses the class into separate files is considered good design.

Using Separate Files

- Place class declaration in a header file that serves as the class specification file. Name the file ***`classname.h`*** (for example, **`Square.h`**)
- Place member function definitions in a class implementation file. Name the file ***`classname.cpp`*** (for example, **`Square.cpp`**) This file should **`#include`** the class specification file.
- A client program (client code) that uses the class must **`#include`** the class specification file and be compiled and linked with the class implementation file.

Include Guards

- Are used to prevent a header file from being included twice
- Format:

```
#ifndef symbol_name
#define symbol_name
. . .    (normal contents of header file)
#endif
```

- *`symbol_name`* is usually the name of the header file, in all capital letters:

```
#ifndef SQUARE_H
#define SQUARE_H
. . .
#endif
```

What Should Be Done Inside *vs.* Outside of the Class

- Class should be designed to provide functions to store and retrieve data
- In general, input and output (I/O) should be done by functions that use class objects, rather than by class member functions

7.12 Structures

- Structure: A programmer-defined data type that allows multiple variables to be grouped together
- Structure declaration format:

```
struct structure name
{
 type1 field1;
 type2 field2;
    …
 typen fieldn;
};
```

Example struct Declaration

```
struct Student  // structure name
{
  int studentID;// structure
  string name;  // members
  short year;
  double gpa;
};              // The ; is required
```

`struct` Declaration Notes

- `struct` names commonly begin with an uppercase letter
- Multiple fields of same type can be declared in a comma-separated list

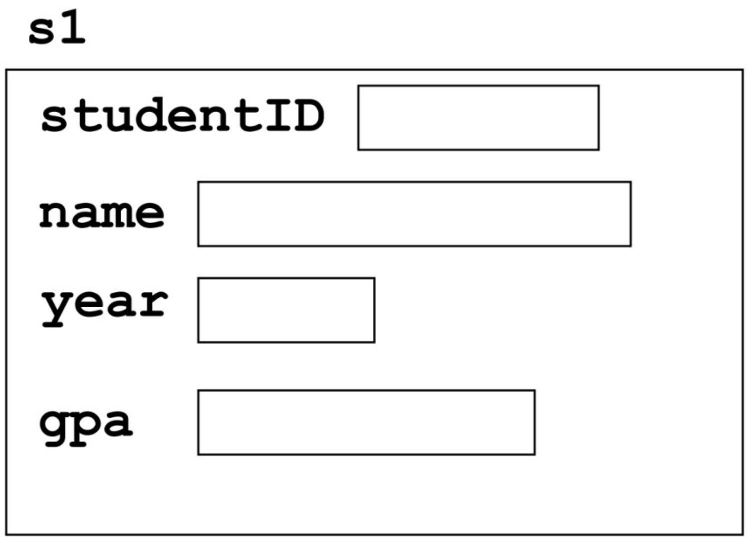
```
   string name,
          address;
```

- Fields in a structure are all public by default

Defining Structure Variables

- A `struct` declaration does not allocate memory or create variables
- To define variables, use the structure name as the type name

```
    Student s1;
```



Accessing Structure Members

- Use the dot `(.)` operator to refer to the data members of **`struct`** variables

```
getline(cin, s1.name);
cin >> s1.studentID;
s1.gpa = 3.75;
```

- The member variables can be used in any manner appropriate for their data type

Displaying `struct` Members

To display the contents of a **`struct`** variable, you must display each field separately, using the dot operator

Wrong:

```
cout << s1; // won't work!
```

Correct:

```
cout << s1.studentID << endl;
cout << s1.name << endl;
cout << s1.year << endl;
cout << s1.gpa;
```

Comparing `struct` Members

- Similar to displaying a **`struct`**, you cannot compare two **`struct`** variables directly:

```
if (s1 >= s2) // won't work!
```

- Instead, compare member variables:

```
if (s1.gpa >= s2.gpa) // better
```

Initializing a Structure 1 of 2

Structure members cannot be initialized in the structure declaration, because no memory has been allocated yet

```
struct Student          // Illegal
{                       // initialization
   int studentID = 1145;
   string name = "Alex";
   short year = 1;
   float gpa = 2.95;
};
```

Pearson

Initializing a Structure 2 of 2

- Structure members are initialized at the time a structure variable is created
- You can initialize a structure variable’s members with either
  - an initialization list, or
  - a constructor

Using an Initialization List

An initialization list is an ordered set of values, separated by commas and contained in `{ }`, that provides initial values for a set of data members

```
{12, 6, 3}  // initialization list
            // with 3 values
```

More on Initialization Lists

- The order of list elements matters: The first value initializes first data member, second value initializes second data member, etc.
- The elements of an initialization list can be constants, variables, or expressions

```
{12, W, L/W + 1} // initialization list
                 // with 3 items
```

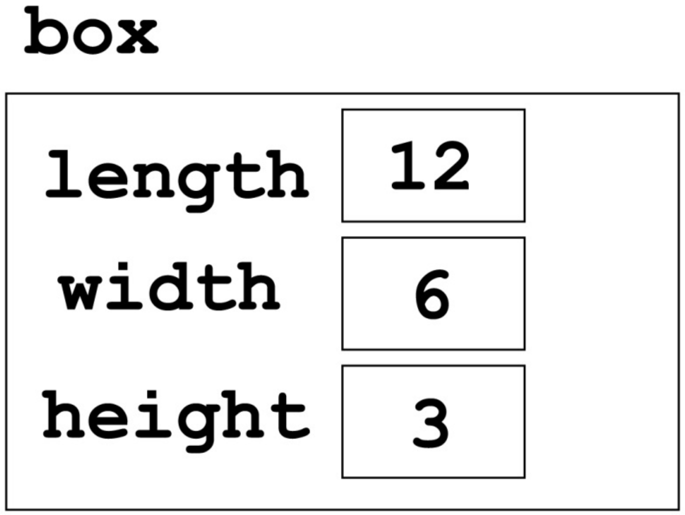
Initialization List Example

Structure Declaration

```
struct Dimensions
{ int length,
      width,
      height;
};

Dimensions box = {12,6,3};
```

Structure Variable



Partial Initialization

You can initialize some of the members, but you cannot skip over members

```
Dimensions box1 = {12,6};  //OK
Dimensions box2 = {12,,3}; //illegal
```

Problems with Using an Initialization List

- You can’t omit a value for a data member without omitting values for all following members
- It does not work on most modern compilers if the structure contains objects, *e.g.*, string objects

Using a Constructor to Initialize Structure Members

- This is similar to a constructor for a class:
  - the name is the same as the name of the struct
  - it has no return type
  - it is used to initialize data members
- It is normally written inside the `struct` declaration

A Structure with a Constructor

```
struct Dimensions
{
   int length,
       width,
       height;
   // Constructor
   Dimensions(int L, int W, int H)
   {length = L; width = W; height = H;}
};
```

Nested Structures

A structure can have another structure as a member.

```
struct PersonInfo
{   string name,
           address,
           city;
};
struct Student
{   int          studentID;
    PersonInfo   pData;
    short        year;
    double       gpa;
};
```

Members of Nested Structures

Use the dot operator multiple times to access fields of nested structures

```
Student s5;
s5.pData.name = "Joanne";
s5.pData.city = "Tulsa";
```

Reference the nested structure's fields by the member variable name, not by the structure name

```
s5.PersonInfo.name = "Joanne"; //no!
```

Structures as Function Arguments

- You may pass members of **`struct`** variables to functions

```
    computeGPA(s1.gpa);
```

- You may pass entire **`struct`** variables to functions

```
    showData(s5);
```

- You can use a reference parameter if the function needs to modify the contents of the structure variable

Notes on Passing Structures

- Using a value parameter for structure can slow down a program and waste space
- Using a reference parameter speeds up program execution, but it allows the function to modify data in the structure
- To save space and time while protecting structure data that should not be changed, use a `const` reference parameter

```
void showData(const Student &s)
                         // header
```

Returning a Structure from a Function

- A function can return a **`struct`**

```
Student getStuData();   // prototype
s1 = getStuData();      // call
```

- The function must define a local structure variable
  - for internal use
  - to use with **`return`** statement

Returning a Structure Example

```
Student getStuData()
{ Student s;    // local variable
  cin >> s.studentID;
  cin.ignore();
  getline(cin, s.pData.name);
  getline(cin, s.pData.address);
  getline(cin, s.pData.city);
  cin >> s.year;
  cin >> s.gpa;
  return s;
}
```

7.13 More About Enumerated Data Types

Additional ways that enumerated data types can be used:

- Data type declaration and variable definition in a single statement:

```
enum Tree {ASH, ELM, OAK} tree1, tree2;
```

- Assign an int value to an enum variable:

```
enum Tree {ASH, ELM, OAK} tree1;
tree1 = static_cast<Tree>(1);  // ELM
```

More About Enumerated Data Types 1 of 4

- Assign the value of an enum variable to an int:

```
enum Tree {ASH, ELM, OAK} tree1;
tree1 = ELM;
int thisTree = tree1;   // assigns 1
int thatTree = OAK;     // assigns 2
```

More About Enumerated Data Types 2 of 4

- Assign the result of a computation to an enum variable

```
enum Tree {ASH, ELM, OAK} tree1, tree2;
tree1 = ELM;
tree2 = static_cast<Tree>(tree1 + 1);
                            // assigns OAK
```

More About Enumerated Data Types 3 of 4

- Using enumerators in a switch statement

```
enum Tree {ASH, ELM, OAK} tree1;
switch(tree1)
{
 case ASH: cout << "Ash";
           break;
 case ELM: cout << "Elm";
           break;
 case OAK: cout << "Oak";
           break;
}
```

More About Enumerated Data Types 4 of 4

- Using enumerators for loop control:

```
enum Tree {ASH, ELM, OAK};
for (Tree tree1 = ASH; tree1 <= OAK;
         tree1=static_cast<Tree>(tree1+1))
{
  .
  .
  .
}
```

Strongly Typed enums (C++ 11) 1 of 4

- Enumerated values (names) cannot be re-used in different enumerated data types that are in the same scope.
- A strongly typed enum (an enum class) allows you to do this

```
enum class Tree {ASH, ELM, OAK};
enum class Street {RUSH, OAK, STATE};
```

- Note the keyword **`class`** in the declaration

Strongly Typed enums (C++ 11) 2 of 4

- Because enumerators can be used in multiple enumerated data types, references must include the name of the strongly typed enum followed by : :

```
enum class Tree : char {ASH, ELM, OAK};

Tree tree1 = Trees::OAK;
```

Strongly Typed enums (C++ 11) 3 of 4

- Strongly typed enumerators are stored as ints by default.
- To choose a different integer data type, indicate the type after the enum name and before the enumerator list:

```
enum class Tree : char {ASH, ELM, OAK};
```

Strongly Typed enums (C++ 11) 4 of 4

- To retrieve the integer value associated with a strongly-typed enumerator, cast it to an int:

```
enum class Tree : char {ASH, ELM, OAK};
int val = static_cast<int>(Trees::ASH);
```

7.15 Introduction to Object-Oriented Analysis and Design

- Object-Oriented Analysis: that phase of program development when the program functionality is determined from the requirements
- It includes
  - identification of classes and objects
  - definition of each class's attributes
  - definition of each class's behaviors
  - definition of the relationship between classes

Identify Classes and Objects

- Consider the major data elements and the operations on these elements
- Candidates include
  - user-interface components (menus, text boxes, *etc.*)
  - I/O devices
  - physical objects
  - historical data (employee records, transaction logs, *etc.*)
  - the roles of human participants

Define Class Attributes

- Attributes are the data elements of an object of the class
- They are necessary for the object to work in its role in the program

Define Class Behaviors

- For each class,
  - Identify what an object of a class should do in the program
- The behaviors determine some of the member functions of the class

Relationships Between Classes

Possible relationships

- Access ("uses-a")
- Ownership/Composition ("has-a")
- Inheritance ("is-a")

Finding the Classes

Technique:

- Write a description of the problem domain (objects, events, etc. related to the problem)
- List the nouns, noun phrases, and pronouns. These are all candidate objects
- Refine the list to include only those objects that are applicable to the problem

Determine Class Responsibilities

Class responsibilities:

- What is the class responsible to know?
- What is the class responsible to do?

Use these to define some of the member functions

Object Reuse

- A well-defined class can be used to create objects in multiple programs
- By re-using an object definition, program development time is shortened
- One goal of object-oriented programming is to support object reuse

Object-Based vs. Object-Oriented

- A program that uses classes and objects is object-based.
- An object-based program that
  - defines relationships between classes of objects
  - creates classes from other classes
  - determines member function behavior based on the object

is more likely to be an object-oriented program

7.16 Screen Control

- Programs to date have all displayed output starting at the upper left corner of the computer screen or output window. Output is displayed left-to-right, line-by-line.
- Computer operating systems are designed to allow programs to access any part of the computer screen. Such access is operating system-specific.

Screen Control – Concepts

- An output screen can be thought of as a grid of 25 rows and 80 columns. Row 0 is at the top of the screen. Column 0 is at the left edge of the screen.
- The intersection of a row and a column is a cell. It can display a single character.
- A cell is identified by its row and column number. These are its coordinates.

Screen Control – Windows - Specific

- `#include <windows.h>` to access the operating system from a program
- Create a handle to reference the output screen:

```
    HANDLE screen =
    GetStdHandle(STD_OUTPUT_HANDLE);
```

- Create a COORD structure to hold the coordinates of a cell on the screen:

```
     COORD position;
```

Screen Control – Windows – More Specifics

- Assign coordinates where the output should appear:

```
position.X = 30;     // column
position.Y = 12;     // row
```

- Set the screen cursor to this cell:

```
SetConsoleCursorPosition(screen,
 position);
```

- Send output to the screen:

```
cout << "Look at me!" << endl;
```

  – be sure to end with **`endl`**, not **`'\n'`** or nothing

Copyright