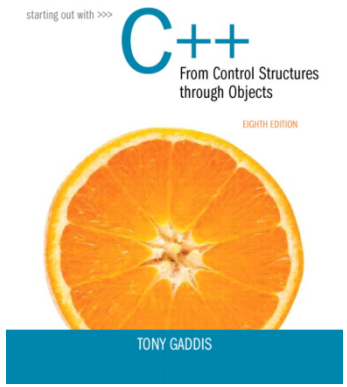
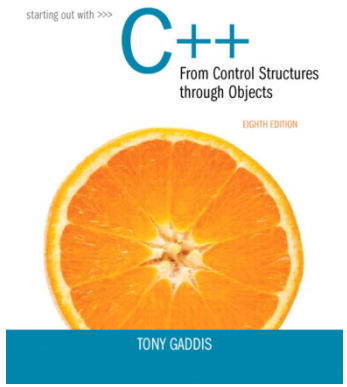


CPSC 5011: Object-Oriented Concepts

Lecture 0B: Advanced C++

Gaddis, Chapters 9, 11, 13, 14

Addison-Wesley
is an imprint of
PEARSON



13.1

Procedural and Object-Oriented Programming

Addison-Wesley
is an imprint of
PEARSON

Procedural and Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program

- 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

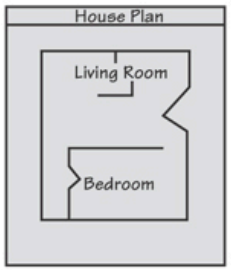
Object-Oriented Programming Terminology

- class: like a `struct` (allows bundling of related variables), but variables and functions in the class can have different properties than in a `struct`
- object: an instance of a `class`, in the same way that a variable can be an instance of a `struct`

Classes and Objects

- A Class is like a blueprint and objects are like houses built from the blueprint

Blueprint that describes a house.



Instances of the house described by the blueprint.

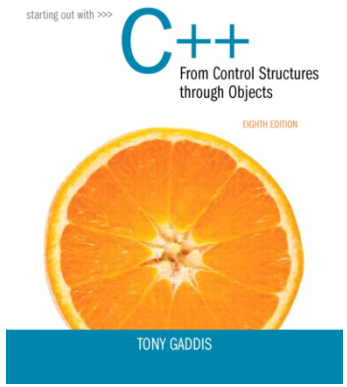
Object-Oriented Programming Terminology

- attributes: members of a class
- methods or behaviors: member functions of a class

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



13.2

Introduction to Classes

Addison-Wesley
is an imprint of
PEARSON

Introduction to Classes

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

```
class ClassName
{
        declaration;
        declaration;
};
```

Class Example

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

Access Specifiers

- Used to control access to members of the class
- `Public:` can be accessed by functions outside of the class
- `private:` can only be called by or accessed by functions that are members of the class

Class Example

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

Private Members

Public Members

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`

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;
```

Defining a Member Function

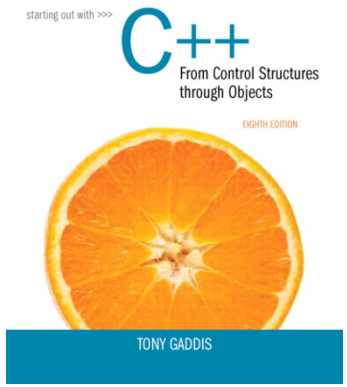
- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator `(::)`

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

Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way

- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.



13.3

Defining an Instance of a Class

Addison-Wesley
is an imprint of
PEARSON

Defining an Instance of a Class

- An object is an instance of a class
- Defined like structure variables:

```
Rectangle r;
```

- Access members using dot operator:

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

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

Program 13-1

```
1  // This program demonstrates a simple class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration.
6  class Rectangle
7  {
8     private:
9        double width;
10       double length;
11    public:
12       void setWidth(double);
13       void setLength(double);
14       double getWidth() const;
15       double getLength() const;
16       double getArea() const;
17 };
18
19 //***************************************************
20 // setWidth assigns a value to the width member.   *
21 //***************************************************
22
23 void Rectangle::setWidth(double w)
24 {
25    width = w;
26 }
27
28 //***************************************************
29 // setLength assigns a value to the length member. *
30 //***************************************************
31
```

Program 13-1 (Continued)

```
32  void Rectangle::setLength(double len)
33  {
34     length = len;
35  }
36
37  //*****************************************************
38  // getWidth returns the value in the width member. *
39  //*****************************************************
40
41  double Rectangle::getWidth() const
42  {
43     return width;
44  }
45
46  //*****************************************************
47  // getLength returns the value in the length member. *
48  //*****************************************************
49
50  double Rectangle::getLength() const
51  {
52     return length;
53  }
54
```

Program 13-1 (Continued)

```
55  //*******************************************************
56  // getArea returns the product of width times length. *
57  //*******************************************************
58
59  double Rectangle::getArea() const
60  {
61     return width * length;
62  }
63
64  //*******************************************************
65  // Function main                                       *
66  //*******************************************************
67
68  int main()
69  {
70     Rectangle box;     // Define an instance of the Rectangle class
71     double rectWidth;  // Local variable for width
72     double rectLength; // Local variable for length
73
74     // Get the rectangle's width and length from the user.
75     cout << "This program will calculate the area of a\n";
76     cout << "rectangle. What is the width? ";
77     cin >> rectWidth;
78     cout << "What is the length? ";
79     cin >> rectLength;
80
81     // Store the width and length of the rectangle
82     // in the box object.
83     box.setWidth(rectWidth);
84     box.setLength(rectLength);
```

Program 13-1 (Continued)

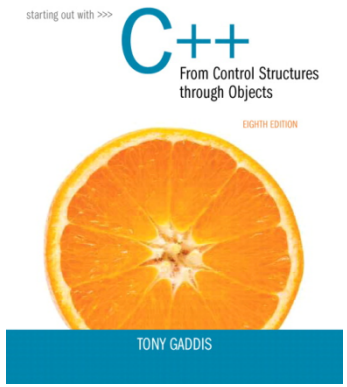
```
85
86      // Display the rectangle's data.
87      cout << "Here is the rectangle's data:\n";
88      cout << "Width: " << box.getWidth() << endl;
89      cout << "Length: " << box.getLength() << endl;
90      cout << "Area: " << box.getArea() << endl;
91      return 0;
92  }
```

Program Output

```
This program will calculate the area of a
rectangle. What is the width? 10 [Enter]
What is the length? 5 [Enter]
Here is the rectangle's data:
Width: 10
Length: 5
Area: 50
```

Avoiding Stale Data

- Some data is the result of a calculation.
- In the `Rectangle` class the area of a rectangle is calculated.
  - length x width
- If we were to use an `area` variable here in the `Rectangle` class, its value would be dependent on the length and the width.
- If we change `length` or `width` without updating `area`, then `area` would become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.



13.4

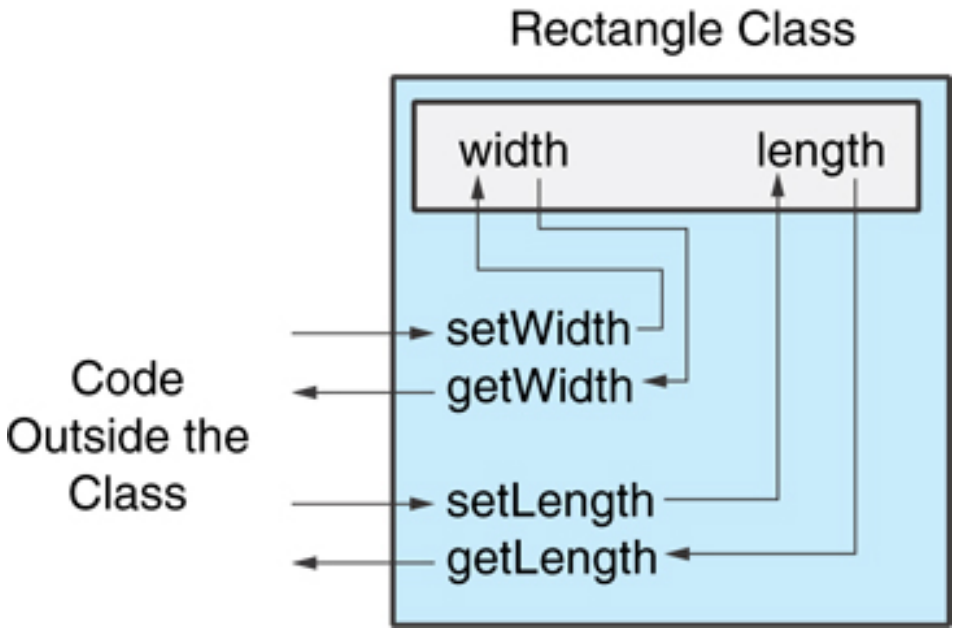
Why Have Private Members?

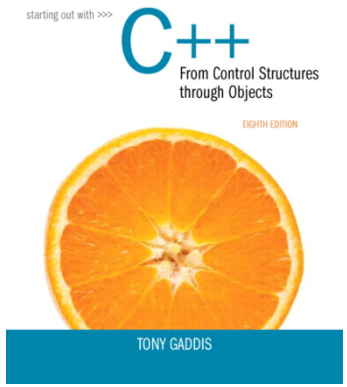
Addison-Wesley
is an imprint of
PEARSON

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.





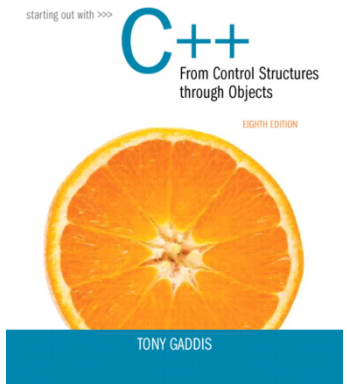
13.5

Separating Specification from Implementation

Addison-Wesley
is an imprint of
PEARSON

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`
- Place member function definitions in *ClassName*`.cpp`, for example, `Rectangle.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



13.7

Constructors

Addison-Wesley
is an imprint of
PEARSON

Constructors

- Member function that is automatically called when an object is created
- Purpose is to construct an object
- Constructor function name is class name
- Has no return type

Contents of `Rectangle.h` (Version 3)

```
1  // Specification file for the Rectangle class
2  // This version has a constructor.
3  #ifndef RECTANGLE_H
4  #define RECTANGLE_H
5
6  class Rectangle
7  {
8     private:
9        double width;
10       double length;
11    public:
12       Rectangle();              // Constructor
13       void setWidth(double);
14       void setLength(double);
15
16       double getWidth() const
17          { return width; }
18
19       double getLength() const
20          { return length; }
21
22       double getArea() const
23          { return width * length; }
24 };
25 #endif
```

Contents of `Rectangle.cpp` (Version 3)

```
1  // Implementation file for the Rectangle class.
2  // This version has a constructor.
3  #include "Rectangle.h"   // Needed for the Rectangle class
4  #include <iostream>      // Needed for cout
5  #include <cstdlib>       // Needed for the exit function
6  using namespace std;
7
8  //*********************************************************
9  // The constructor initializes width and length to 0.0.   *
10 //*********************************************************
11
12 Rectangle::Rectangle()
13 {
14    width = 0.0;
15    length = 0.0;
16 }
```

Continues...

Contents of Rectangle.cpp Version 3 (Continued)

```
17
18  //*************************************************************
19  // setWidth sets the value of the member variable width.    *
20  //*************************************************************
21
22  void Rectangle::setWidth(double w)
23  {
24     if (w >= 0)
25        width = w;
26     else
27     {
28        cout << "Invalid width\n";
29        exit(EXIT_FAILURE);
30     }
31  }
32
33  //*************************************************************
34  // setLength sets the value of the member variable length.  *
35  //*************************************************************
36
37  void Rectangle::setLength(double len)
38  {
39     if (len >= 0)
40        length = len;
41     else
42     {
43        cout << "Invalid length\n";
44        exit(EXIT_FAILURE);
45     }
46  }
```

(continued)

Program 13-7

```
1  // This program uses the Rectangle class's constructor.
2  #include <iostream>
3  #include "Rectangle.h" // Needed for Rectangle class
4  using namespace std;
5
6  int main()
7  {
8      Rectangle box;     // Define an instance of the Rectangle class
9
10     // Display the rectangle's data.
11     cout << "Here is the rectangle's data:\n";
12     cout << "Width: " << box.getWidth() << endl;
13     cout << "Length: " << box.getLength() << endl;
14     cout << "Area: " << box.getArea() << endl;
15     return 0;
16 }
```

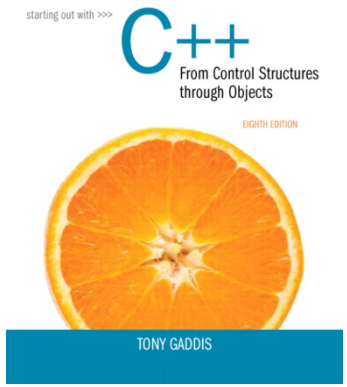
Program Output

```
Here is the rectangle's data:
Width: 0
Length: 0
Area: 0
```

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.
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```



13.8

Passing Arguments to Constructors

Addison-Wesley
is an imprint of
PEARSON

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;
}
```

Passing Arguments to Constructors

- You can pass arguments to the constructor when you create an object:

```
Rectangle r(10, 5);
```

More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

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

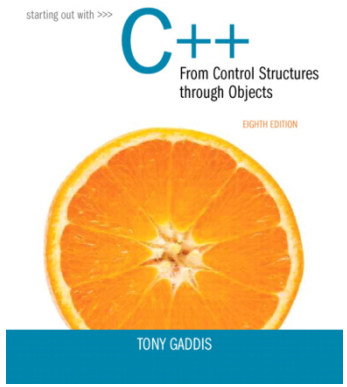
- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.

- When this is the case, you must pass the required arguments to the constructor when creating an object.



13.10

Overloading Constructors

Addison-Wesley
is an imprint of
PEARSON

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);
```

```
1  // This class has overloaded constructors.
2  #ifndef INVENTORYITEM_H
3  #define INVENTORYITEM_H
4  #include <string>
5  using namespace std;
6
7  class InventoryItem
8  {
9  private:
10    string description; // The item description
11    double cost;        // The item cost
12    int units;          // Number of units on hand
13 public:
14    // Constructor #1
15    InventoryItem()
16       { // Initialize description, cost, and units.
17         description = "";
18         cost = 0.0;
19         units = 0; }
20
21    // Constructor #2
22    InventoryItem(string desc)
23       { // Assign the value to description.
24         description = desc;
25
26         // Initialize cost and units.
27         cost = 0.0;
28         units = 0; }
```

Continues...

```
29
30      // Constructor #3
31      InventoryItem(string desc, double c, int u)
32        { // Assign values to description, cost, and units.
33          description = desc;
34          cost = c;
35          units = u; }
36
37      // Mutator functions
38      void setDescription(string d)
39         { description = d; }
40
41      void setCost(double c)
42         { cost = c; }
43
44      void setUnits(int u)
45         { units = u; }
46
47      // Accessor functions
48      string getDescription() const
49         { return description; }
50
51      double getCost() const
52         { return cost; }
53
54      int getUnits() const
55         { return units; }
56  };
57  #endif
```

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
```

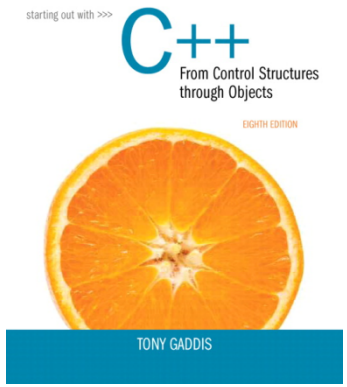
- Since a destructor takes no arguments, there can only be one destructor for a class

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



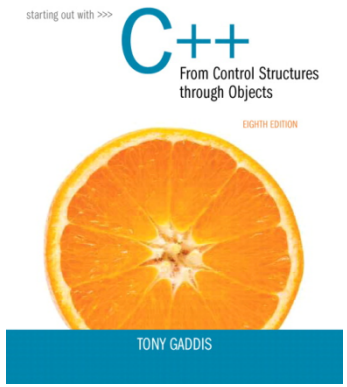
3.11

Using Private Member Functions

Addison-Wesley
is an imprint of
PEARSON

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
- See the `createDescription` function in **`ContactInfo.h`** (Version 2)



13.12

Arrays of Objects

Addison-Wesley
is an imprint of
PEARSON

Arrays of Objects

- Objects can be the elements of an array:

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

- Default constructor for object is used when array is defined

Arrays of Objects

- Must use initializer list to invoke constructor that takes arguments:

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

Arrays of Objects

- 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) };
```

Arrays of Objects

- 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();
```

Program 13-14

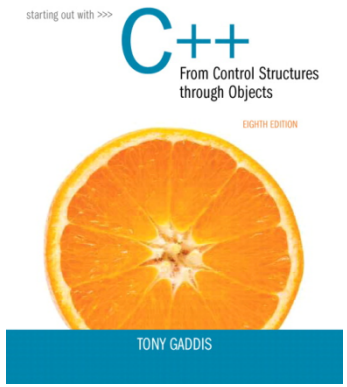
```
1  // This program demonstrates an array of class objects.
2  #include <iostream>
3  #include <iomanip>
4  #include "InventoryItem.h"
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_ITEMS = 5;
10    InventoryItem inventory[NUM_ITEMS] = {
11                  InventoryItem("Hammer", 6.95, 12),
12                  InventoryItem("Wrench", 8.75, 20),
13                  InventoryItem("Pliers", 3.75, 10),
14                  InventoryItem("Ratchet", 7.95, 14),
15                  InventoryItem("Screwdriver", 2.50, 22) };
16
```

Program 13-14 (Continued)

```
17      cout << setw(14) <<"Inventory Item"
18           << setw(8) << "Cost" << setw(8)
19           << setw(16) << "Units on Hand\n";
20      cout << "---------------------------------------\n";
21
22      for (int i = 0; i < NUM_ITEMS; i++)
23      {
24         cout << setw(14) << inventory[i].getDescription();
25         cout << setw(8) << inventory[i].getCost();
26         cout << setw(7) << inventory[i].getUnits() << endl;
27      }
28
29      return 0;
30   }
```

Program Output

```
Inventory Item    Cost   Units on Hand
---------------------------------------
         Hammer    6.95         12
         Wrench    8.75         20
         Pliers    3.75         10
        Ratchet    7.95         14
    Screwdriver     2.5         22
```



9.1

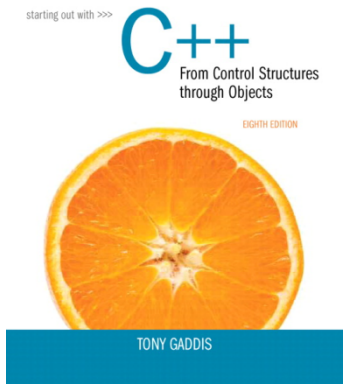
Getting the Address of a Variable

Addison-Wesley
is an imprint of
PEARSON

Getting the Address of a Variable

- Each variable in program is stored at a unique address
- Use address operator `&` to get address of a variable:

```
int num = -99;
cout << &num; // prints address
              // in hexadecimal
```



9.2

Pointer Variables

Addison-Wesley
is an imprint of
PEARSON

Pointer Variables

- Pointer variable : Often just called a pointer, it's a variable that holds an address

- Because a pointer variable holds the address of another piece of data, it "points" to the data

Something Like Pointers: Arrays

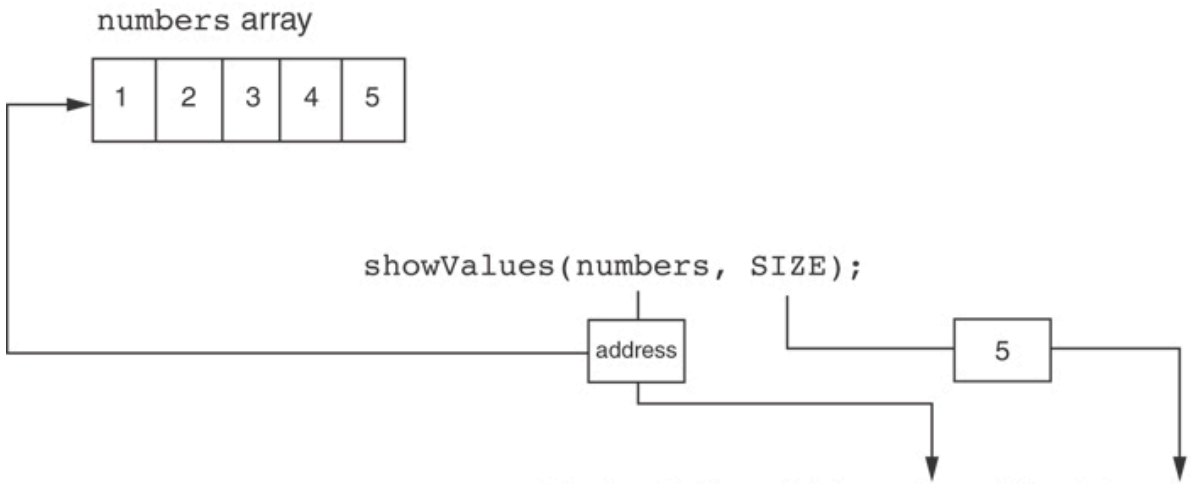
- We have already worked with something similar to pointers, when we learned to pass arrays as arguments to functions.

- For example, suppose we use this statement to pass the array `numbers` to the `showValues` function:

```
showValues(numbers, SIZE);
```

Something Like Pointers : Arrays

The `values` parameter, in the `showValues` function, points to the `numbers` array.



C++ automatically stores the address of `numbers` in the `values` parameter.

```
void showValues(int values[], int size)
{
   for (int count = 0; count < size; count++)
      cout << values[count] << endl;
}
```

Something Like Pointers: Reference Variables

- We have also worked with something like pointers when we learned to use reference variables. Suppose we have this function:

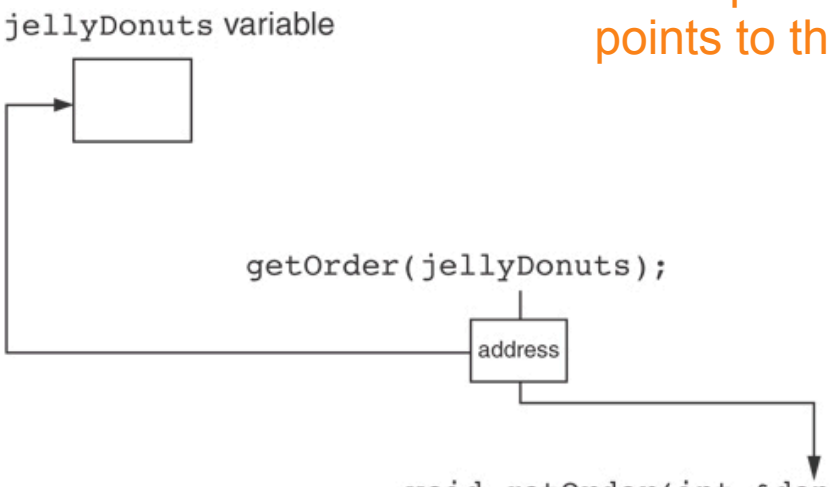
```
void getOrder(int &donuts)
{
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

- And we call it with this code:

```
int jellyDonuts;
getOrder(jellyDonuts);
```

Something Like Pointers: Reference Variables

The `donuts` parameter, in the `getOrder` function, points to the `jellyDonuts` variable.




```
void getOrder(int &donuts)
{
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

C++ automatically stores the address of `jellyDonuts` in the `donuts` parameter.

Pointer Variables

- Pointer variables are yet another way using a memory address to work with a piece of data.
- Pointers are more "low-level" than arrays and reference variables.
- This means you are responsible for finding the address you want to store in the pointer and correctly using it.

Pointer Variables

- Definition:

```
int  *intptr;
```

- Read as:

"`intptr` can hold the address of an int"

- Spacing in definition does not matter:

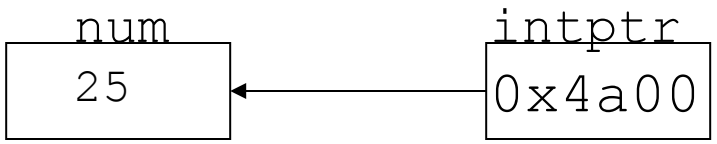
```
int * intptr;  // same as above
int*  intptr;  // same as above
```

Pointer Variables

- Assigning an address to a pointer variable:

```
int *intptr;
intptr = &num;
```

- Memory layout:



address of `num`: `0x4a00`

Pointer Variables

- Initialize pointer variables with the special value `nullptr`.
- In C++ 11, the `nullptr` key word was introduced to represent the address `0`.
- Here is an example of how you define a pointer variable and initialize it with the value `nullptr`:

```
int *ptr = nullptr;
```

A Pointer Variable in Program 9-2

Program 9-2

```
1  // This program stores the address of a variable in a pointer.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int x = 25;           // int variable
8       int *ptr = nullptr;   // Pointer variable, can point to an int
9
10      ptr = &x;             // Store the address of x in ptr
11      cout << "The value in x is " << x << endl;
12      cout << "The address of x is " << ptr << endl;
13      return 0;
14 }
```

Program Output

```
The value in x is 25
The address of x is 0x7e00
```

The Indirection Operator

- The indirection operator (*) dereferences a pointer.
- It allows you to access the item that the pointer points to.

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

This prints 25.

The Indirection Operator in Program 9-3

Program 9-3

```
1  // This program demonstrates the use of the indirection operator.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int x = 25;           // int variable
8       int *ptr = nullptr;   // Pointer variable, can point to an int
9
10      ptr = &x;             // Store the address of x in ptr
11
12      // Use both x and ptr to display the value in x.
13      cout << "Here is the value in x, printed twice:\n";
14      cout << x << endl;    // Displays the contents of x
15      cout << *ptr << endl; // Displays the contents of x
16
17      // Assign 100 to the location pointed to by ptr. This
18      // will actually assign 100 to x.
19      *ptr = 100;
```

(program continues)

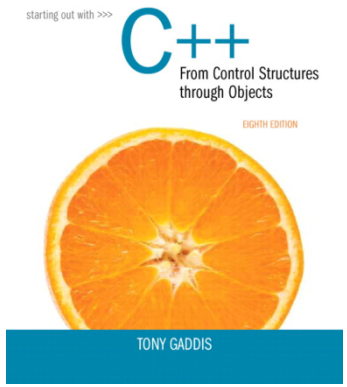
The Indirection Operator in Program 9-3

Program 9-3 *(continued)*

```
20
21      // Use both x and ptr to display the value in x.
22      cout << "Once again, here is the value in x:\n";
23      cout << x << endl;     // Displays the contents of x
24      cout << *ptr << endl; // Displays the contents of x
25      return 0;
26  }
```

Program Output

```
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
```



9.3

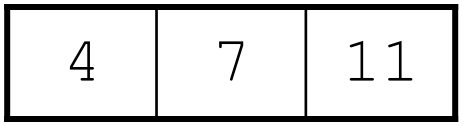
The Relationship Between Arrays and Pointers

Addison-Wesley
is an imprint of
PEARSON

The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```



starting address of `vals`: `0x4a00`

```
cout << vals;              // displays
                           // 0x4a00
cout << vals[0];       // displays 4
```

The Relationship Between Arrays and Pointers

- Array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

The Array Name Being Dereferenced in Program 9-5

Program 9-5

```
1  // This program shows an array name being dereferenced with the *
2  // operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     short numbers[] = {10, 20, 30, 40, 50};
9
10    cout << "The first element of the array is ";
11    cout << *numbers << endl;
12    return 0;
13 }
```

Program Output

```
The first element of the array is 10
```

Pointers in Expressions

Given:

```
int vals[]={4,7,11}, *valptr;
valptr = vals;
```

What is `valptr + 1`? It means (address in `valptr`) + (1 * size of an int)

```
cout << *(valptr+1); //displays 7
cout << *(valptr+2); //displays 11
```

Must use `( )` as shown in the expressions

Array Access

- Array elements can be accessed in many ways:

Array access method	Example
array name and `[]`	`vals[2] = 17;`
pointer to array and `[]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals + 2) = 17;`
pointer to array and subscript arithmetic	`*(valptr + 2) = 17;`

Array Access

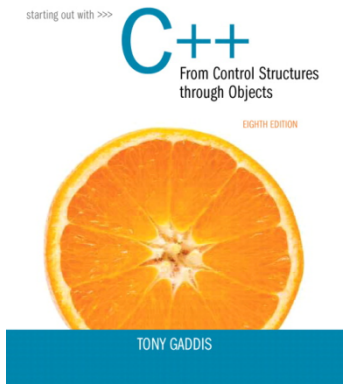
- Conversion: `vals[i]` is equivalent to `*(vals + i)`
- No bounds checking performed on array access, whether using array name or a pointer

From Program 9-7

```
 9     const int NUM_COINS = 5;
10     double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
11     double *doublePtr;   // Pointer to a double
12     int count;           // Array index
13
14     // Assign the address of the coins array to doublePtr.
15     doublePtr = coins;
16
17     // Display the contents of the coins array. Use subscripts
18     // with the pointer!
19     cout << "Here are the values in the coins array:\n";
20     for (count = 0; count < NUM_COINS; count++)
21        cout << doublePtr[count] << " ";
22
23     // Display the contents of the array again, but this time
24     // use pointer notation with the array name!
25     cout << "\nAnd here they are again:\n";
26     for (count = 0; count < NUM_COINS; count++)
27        cout << *(coins + count) << " ";
28     cout << endl;
```

Program Output

```
Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
And here they are again:
0.05 0.1 0.25 0.5 1
```



9.4

Pointer Arithmetic

Pointer Arithmetic

- Operations on pointer variables:

Operation	Example
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;  // points at 11`

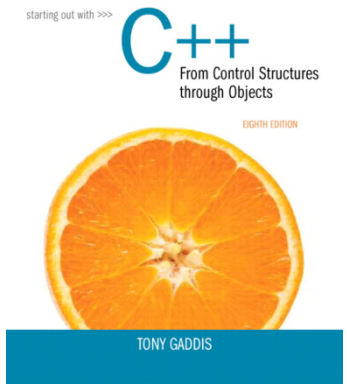
`-` (pointer from pointer)	`cout << valptr-val; // difference`
	`//(number of ints) between valptr`
	`// and val`

From Program 9-9

```
 7       const int SIZE = 8;
 8       int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
 9       int *numPtr = nullptr;  // Pointer
10       int count;              // Counter variable for loops
11
12       // Make numPtr point to the set array.
13       numPtr = set;
14
15       // Use the pointer to display the array contents.
16       cout << "The numbers in set are:\n";
17       for (count = 0; count < SIZE; count++)
18       {
19            cout << *numPtr << " ";
20            numPtr++;
21       }
22
23       // Display the array contents in reverse order.
24       cout << "\nThe numbers in set backward are:\n";
25       for (count = 0; count < SIZE; count++)
26       {
27            numPtr--;
28            cout << *numPtr << " ";
29       }
30       return 0;
31  }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```



9.5

Initializing Pointers

Addison-Wesley
is an imprint of
PEARSON

Initializing Pointers

- Can initialize at definition time:

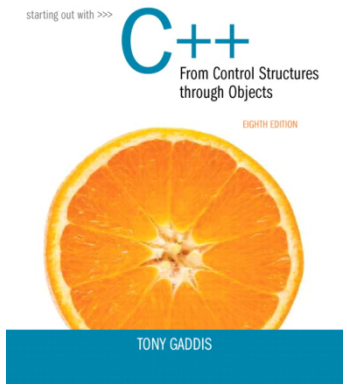
```
int num, *numptr = &num;
int val[3], *valptr = val;
```

- Cannot mix data types:

```
double cost;
int *ptr = &cost; // won't work
```

- Can test for an invalid address for `ptr` with:

```
if (!ptr) ...
```



9.6

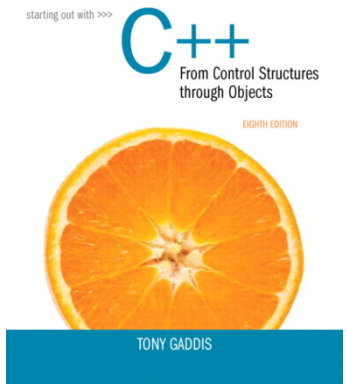
Comparing Pointers

Addison-Wesley
is an imprint of
PEARSON

Comparing Pointers

- Relational operators (<, >=, etc.) can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2)  // compares
                     // contents
```



9.7

Pointers as Function Parameters

Pointers as Function Parameters

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:

1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr); // ptr is pointer to an int
```

2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

3) address as argument to the function

```
getNum(&num);      // pass address of num to getNum
```

Example

```
void swap(int *x, int *y)
{     int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}

int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

Pointers as Function Parameters in Program 9-11

Program 9-11

```
1  // This program uses two functions that accept addresses of
2  // variables as arguments.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void getNumber(int *);
8  void doubleValue(int *);
9
10 int main()
11 {
12    int number;
13
14    // Call getNumber and pass the address of number.
15    getNumber(&number);
16
17    // Call doubleValue and pass the address of number.
18    doubleValue(&number);
19
20    // Display the value in number.
21    cout << "That value doubled is " << number << endl;
22    return 0;
23 }
24
```

(Program Continues)

Pointers as Function Parameters in Program 9-11

Program 9-11 *(continued)*

```
25  //*************************************************************
26  // Definition of getNumber. The parameter, input, is a pointer. *
27  // This function asks the user for a number. The value entered  *
28  // is stored in the variable pointed to by input.               *
29  //*************************************************************
30
31  void getNumber(int *input)
32  {
33     cout << "Enter an integer number: ";
34     cin >> *input;
35  }
36
37  //*************************************************************
38  // Definition of doubleValue. The parameter, val, is a pointer. *
39  // This function multiplies the variable pointed to by val by   *
40  // two.                                                         *
41  //*************************************************************
42
43  void doubleValue(int *val)
44  {
45     *val *= 2;
46  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer number: 10 [Enter]
That value doubled is 20
```

Pointers to Constants

- If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-const.

Pointers to Constants

- Example: Suppose we have the following definitions:

```
const int SIZE = 6;
const double payRates[SIZE] =
      { 18.55, 17.45, 12.85,
        14.97, 10.35, 18.89 };
```

- In this code, `payRates` is an array of constant doubles.

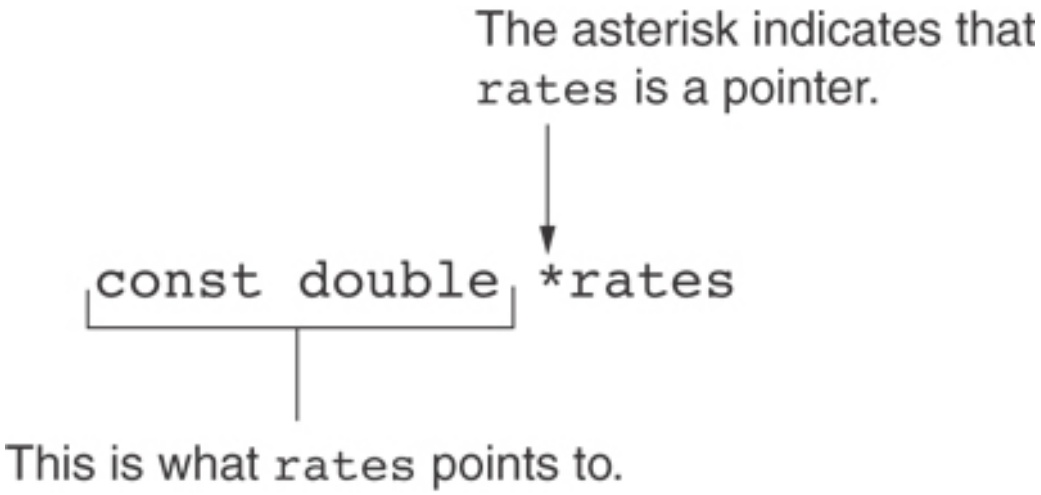
Pointers to Constants

- Suppose we wish to pass the `payRates` array to a function? Here's an example of how we can do it.

```
void displayPayRates(const double *rates, int size)
{
   for (int count = 0; count < size; count++)
   {
      cout << "Pay rate for employee " << (count + 1)
           << " is $" << *(rates + count) << endl;
   }
}
```

The parameter, rates, is a pointer to `const double`.

Declaration of a Pointer to Constant



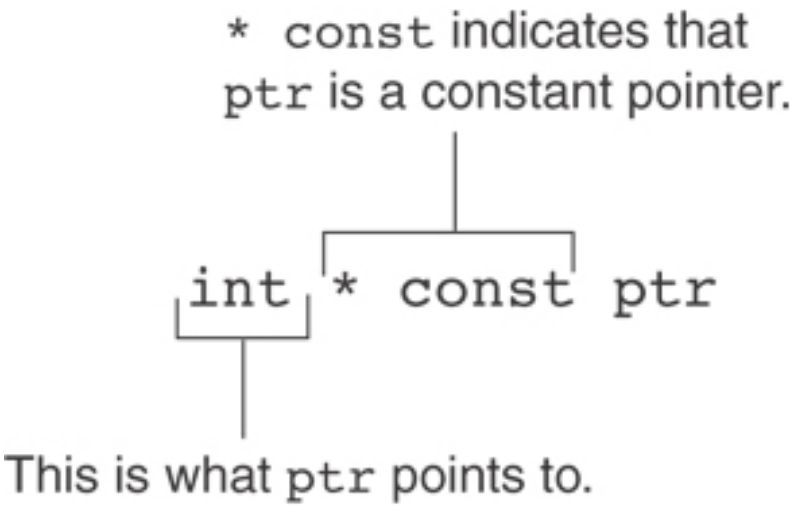
Constant Pointers

- A constant pointer is a pointer that is initialized with an address, and cannot point to anything else.

- Example

```
int value = 22;
int * const ptr = &value;
```

Constant Pointers



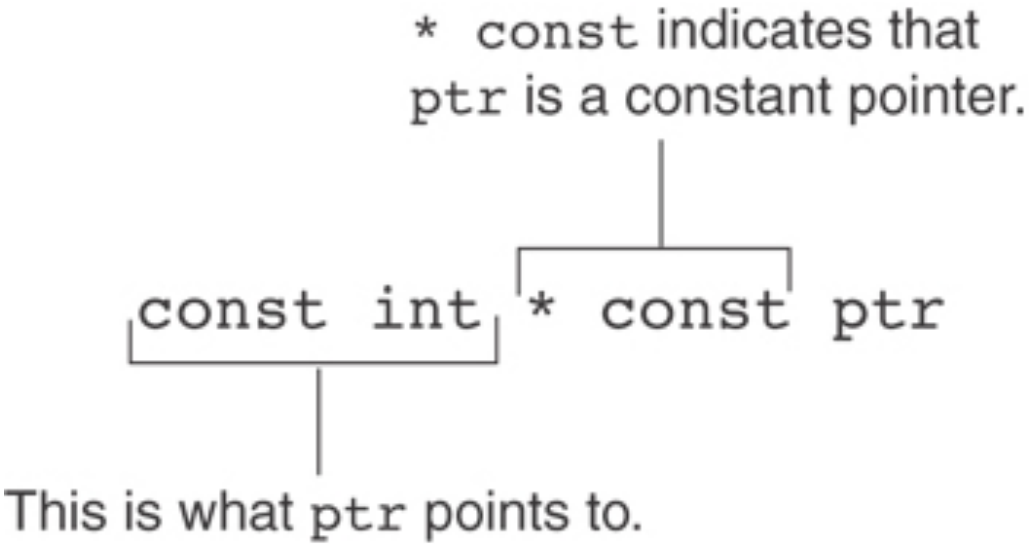
Constant Pointers to Constants

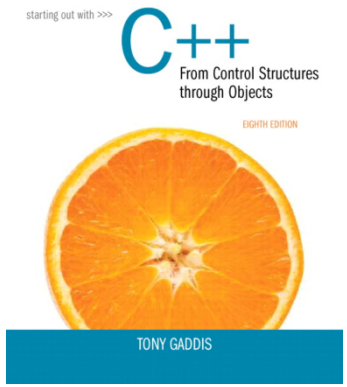
- A constant pointer to a constant is:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to

- Example:

```
int value = 22;
const int * const ptr = &value;
```

Constant Pointers to Constants





9.8

Dynamic Memory Allocation

Addison-Wesley
is an imprint of
PEARSON

Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Computer returns address of newly allocated variable
- Uses `new` operator to allocate memory:

```
double *dptr = nullptr;
dptr = new double;
```

- `new` returns address of memory location

Dynamic Memory Allocation

- Can also use `new` to allocate array:

```
const int SIZE = 25;
arrayPtr = new double[SIZE];
```

- Can then use `[]` or pointer arithmetic to access array:

```
for(i = 0; i < SIZE; i++)
   *arrayptr[i] = i * i;
```

or

```
for(i = 0; i < SIZE; i++)
   *(arrayptr + i) = i * i;
```

- Program will terminate if not enough memory available to allocate

Releasing Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

Dynamic Memory Allocation in Program 9-14

Program 9-14

```
1  // This program totals and averages the sales figures for any
2  // number of days. The figures are stored in a dynamically
3  // allocated array.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10     double *sales = nullptr, // To dynamically allocate an array
11            total = 0.0,      // Accumulator
12            average;          // To hold average sales
13     int numDays,             // To hold the number of days of sales
14         count;               // Counter variable
15
16     // Get the number of days of sales.
17     cout << "How many days of sales figures do you wish ";
18     cout << "to process? ";
19     cin >> numDays;
```

Dynamic Memory Allocation in Program 9-14

```
20
21      // Dynamically allocate an array large enough to hold
22      // that many days of sales amounts.
23      sales = new double[numDays];
24
25      // Get the sales figures for each day.
26      cout << "Enter the sales figures below.\n";
27      for (count = 0; count < numDays; count++)
28      {
29          cout << "Day " << (count + 1) << ": ";
30          cin >> sales[count];
31      }
32
33      // Calculate the total sales
34      for (count = 0; count < numDays; count++)
35      {
36          total += sales[count];
37      }
38
39      // Calculate the average sales per day
40      average = total / numDays;
41
42      // Display the results
43      cout << fixed << showpoint << setprecision(2);
44      cout << "\n\nTotal Sales: $" << total << endl;
45      cout << "Average Sales: $" << average << endl;
```

Program 9-14 (Continued)

Dynamic Memory Allocation in Program 9-14

Program 9-14 (Continued)

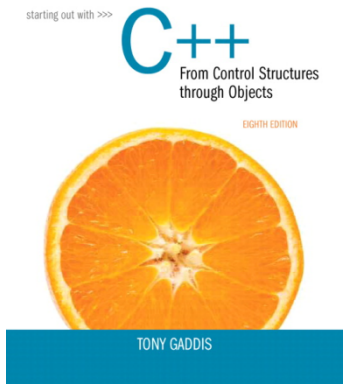
```
46
47      // Free dynamically allocated memory
48      delete [] sales;
49      sales = nullptr;     // Make sales a null pointer.
50
51      return 0;
52  }
```

Program Output with Example Input Shown in Bold

```
How many days of sales figures do you wish to process? 5 [Enter]
Enter the sales figures below.
Day 1: 898.63 [Enter]
Day 2: 652.32 [Enter]
Day 3: 741.85 [Enter]
Day 4: 852.96 [Enter]
Day 5: 921.37 [Enter]

Total Sales: $4067.13
Average Sales: $813.43
```

Notice that in line 49 `nullptr` is assigned to the `sales` pointer. The `delete` operator is designed to have no effect when used on a null pointer.



9.9

Returning Pointers from Functions

Addison-Wesley
is an imprint of
PEARSON

Returning Pointers from Functions

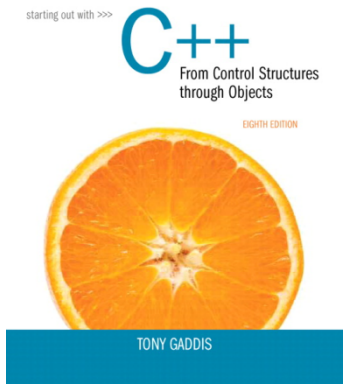
- Pointer can be the return type of a function:

```
int* newNum();
```

- The function must not return a pointer to a local variable in the function.
- A function should only return a pointer:
  - to data that was passed to the function as an argument, or
  - to dynamically allocated memory

From Program 9-15

```
34  int *getRandomNumbers(int num)
35  {
36      int *arr = nullptr;  // Array to hold the numbers
37
38      // Return a null pointer if num is zero or negative.
39      if (num <= 0)
40          return nullptr;
41
42      // Dynamically allocate the array.
43      arr = new int[num];
44
45      // Seed the random number generator by passing
46      // the return value of time(0) to srand.
47      srand( time(0) );
48
49      // Populate the array with random numbers.
50      for (int count = 0; count < num; count++)
51          arr[count] = rand();
52
53      // Return a pointer to the array.
54      return arr;
55  }
```



11.9

Pointers to Structures

Addison-Wesley
is an imprint of
PEARSON

Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:

```
Student *stuPtr;
```

- Can use `&` operator to assign address:

```
stuPtr = & stu1;
```

- Structure pointer can be a function parameter

Accessing Structure Members via Pointer Variables

- Must use `()` to dereference pointer variable, not field within structure:

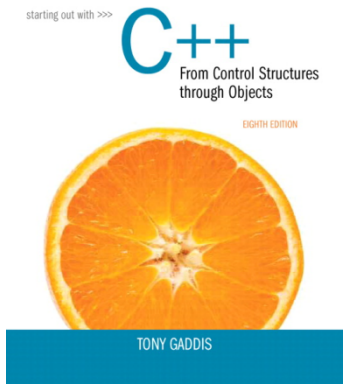
```
cout << (*stuPtr).studentID;
```

- Can use structure pointer operator to eliminate `()` and use clearer notation:

```
cout << stuPtr->studentID;
```

From Program 11-8

```
42  void getData(Student *s)
43  {
44     // Get the student name.
45     cout << "Student name: ";
46     getline(cin, s->name);
47
48     // Get the student ID number.
49     cout << "Student ID Number: ";
50     cin >> s->idNum;
51
52     // Get the credit hours enrolled.
53     cout << "Credit Hours Enrolled: ";
54     cin >> s->creditHours;
55
56     // Get the GPA.
57     cout << "Current GPA: ";
58     cin >> s->gpa;
59  }
```



13.3

Defining an Instance of a Class

Addison-Wesley
is an imprint of
PEARSON

Pointer to an Object

- Can define a pointer to an object:

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

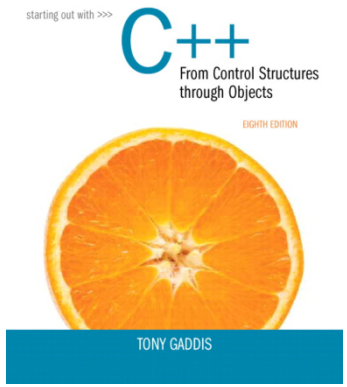
- Can access public members via pointer:

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

Dynamically Allocating an Object

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

```
1  // Define a Rectangle pointer.
2  Rectangle *rectPtr = nullptr;
3
4  // Dynamically allocate a Rectangle object.
5  rectPtr = new Rectangle;
6
7  // Store values in the object's width and length.
8  rectPtr->setWidth(10.0);
9  rectPtr->setLength(15.0);
10
11 // Delete the object from memory.
12 delete rectPtr;
13 rectPtr = nullptr;
```



13.9

Destructors

Addison-Wesley
is an imprint of
PEARSON

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

Contents of `InventoryItem.h` (Version 1)

```
1  // Specification file for the InventoryItem class.
2  #ifndef INVENTORYITEM_H
3  #define INVENTORYITEM_H
4  #include <cstring>   // Needed for strlen and strcpy
5
6  // InventoryItem class declaration.
7  class InventoryItem
8  {
9  private:
10    char *description;  // The item description
11    double cost;        // The item cost
12    int units;          // Number of units on hand
```

Contents of `InventoryItem.h` Version1

```
13  public:
14     // Constructor
15     InventoryItem(char *desc, double c, int u)
16        { // Allocate just enough memory for the description.
17          description = new char [strlen(desc) + 1];
18
19          // Copy the description to the allocated memory.
20          strcpy(description, desc);
21
22          // Assign values to cost and units.
23          cost = c;
24          units = u;}
25
26     // Destructor
27     ~InventoryItem()
28        { delete [] description; }
29
30     const char *getDescription() const
31        { return description; }
32
33     double getCost() const
34        { return cost; }
35
36     int getUnits() const
37        { return units; }
38  };
39  #endif
```

(continued)

Program 13-12

```
1  // This program demonstrates a class with a destructor.
2  #include <iostream>
3  #include "ContactInfo.h"
4  using namespace std;
5
6  int main()
7  {
8       // Define a ContactInfo object with the following data:
9       // Name: Kristen Lee Phone Number: 555-2021
10      ContactInfo entry("Kristen Lee", "555-2021");
11
12      // Display the object's data.
13      cout << "Name: " << entry.getName() << endl;
14      cout << "Phone Number: " << entry.getPhoneNumber() << endl;
15      return 0;
16 }
```

Program Output

```
Name: Kristen Lee
Phone Number: 555-2021
```

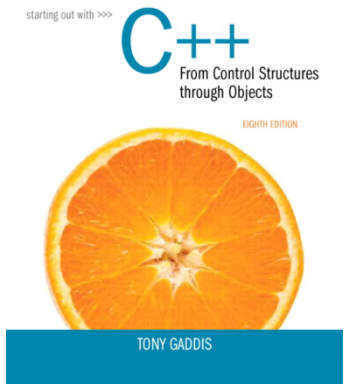
Constructors, Destructors, and Dynamically Allocated Objects

- 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;
```



14.1

Instance and Static Members

Addison-Wesley
is an imprint of
PEARSON

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`

```
 1  // Tree class
 2  class Tree
 3  {
 4  private:
 5     static int objectCount;     // Static member variable.
 6  public:
 7     // Constructor
 8     Tree()
 9        { objectCount++; }
10
11     // Accessor function for objectCount
12     int getObjectCount() const
13        { return objectCount; }
14  };
15
16  // Definition of the static member variable, written
17  // outside the class.
18  int Tree::objectCount = 0;
```

Static member declared here.

Static member defined here.

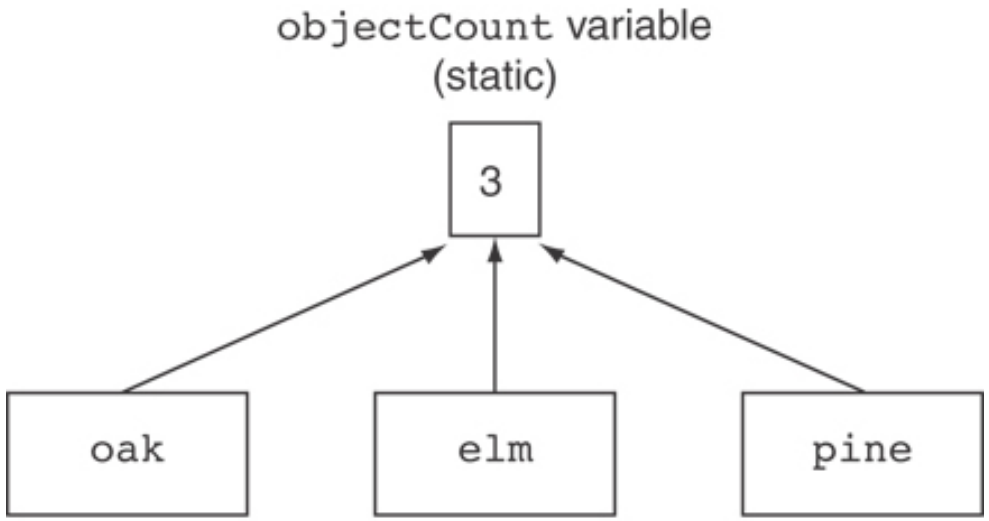
Program 14-1

```
1  // This program demonstrates a static member variable.
2  #include <iostream>
3  #include "Tree.h"
4  using namespace std;
5
6  int main()
7  {
8     // Define three Tree objects.
9     Tree oak;
10    Tree elm;
11    Tree pine;
12
13    // Display the number of Tree objects we have.
14    cout << "We have " << pine.getObjectCount()
15         << " trees in our program!\n";
16    return 0;
17 }
```

Program Output

```
We have 3 trees in our program!
```

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



Instances of the `Tree` class

`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:

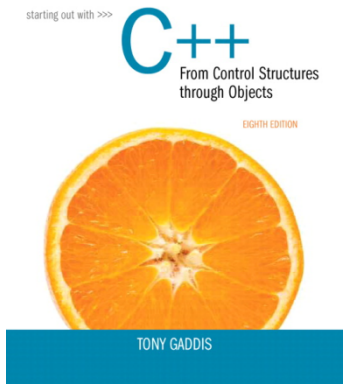
```
int num = Tree::getObjectCount();
```

Modified Version of `Tree.h`

```
 1  // Tree class
 2  class Tree
 3  {
 4  private:
 5     static int objectCount;    // Static member variable.
 6  public:
 7     // Constructor
 8     Tree()
 9        { objectCount++; }
10
11     // Accessor function for objectCount
12     static int getObjectCount() const
13        { return objectCount; }
14  };
15
16  // Definition of the static member variable, written
17  // outside the class.
18  int Tree::objectCount = 0;
```

Now we can call the function like this:

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



14.3

Memberwise Assignment

Addison-Wesley
is an imprint of
PEARSON

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;
```

Program 14-5

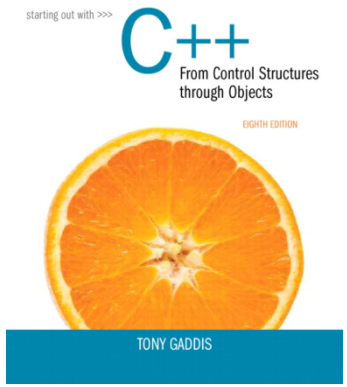
```
1  // This program demonstrates memberwise assignment.
2  #include <iostream>
3  #include "Rectangle.h"
4  using namespace std;
5
6  int main()
7  {
8     // Define two Rectangle objects.
9     Rectangle box1(10.0, 10.0);   // width = 10.0, length = 10.0
10    Rectangle box2 (20.0, 20.0);  // width = 20.0, length = 20.0
11
12    // Display each object's width and length.
13    cout << "box1's width and length: " << box1.getWidth()
14         << " " << box1.getLength() << endl;
15    cout << "box2's width and length: " << box2.getWidth()
16         << " " << box2.getLength() << endl << endl;
17
18    // Assign the members of box1 to box2.
19    box2 = box1;
20
21    // Display each object's width and length again.
22    cout << "box1's width and length: " << box1.getWidth()
23         << " " << box1.getLength() << endl;
24    cout << "box2's width and length: " << box2.getWidth()
25         << " " << box2.getLength() << endl;
26
27    return 0;
28 }
```

Program 14-5 *(continued)*

Program Output

```
box1's width and length: 10 10
box2's width and length: 20 20

box1's width and length: 10 10
box2's width and length: 10 10
```



14.4

Copy Constructors

Addison-Wesley
is an imprint of
PEARSON

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

Copy Constructors

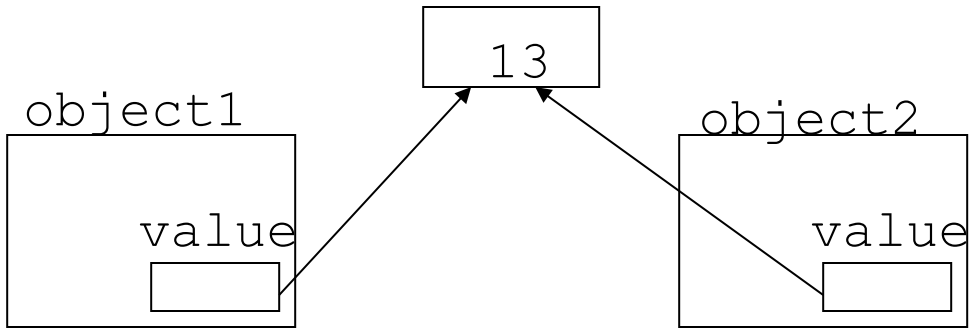
Problem: what if object contains a pointer?

```
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:

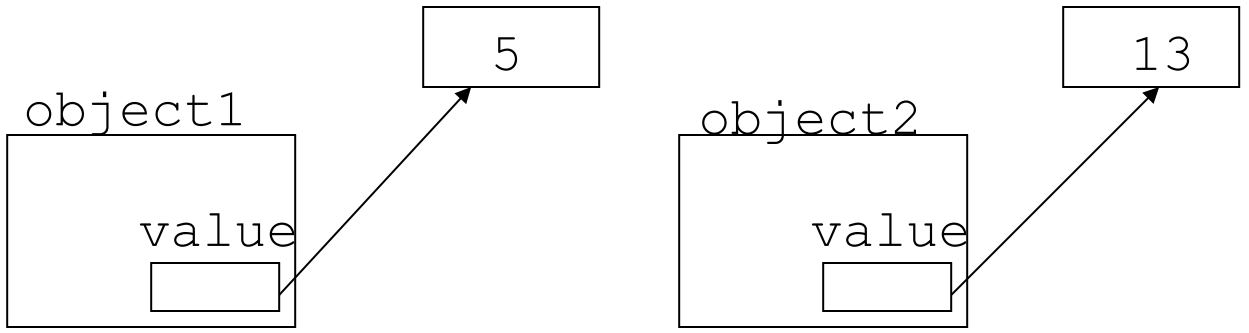
```
SomeClass::SomeClass(const SomeClass &obj)
{
   value = new int;
   *value = obj.value;
}
```

- Copy constructor takes a reference parameter to an object of the class

Programmer-Defined Copy Constructor

- 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`:

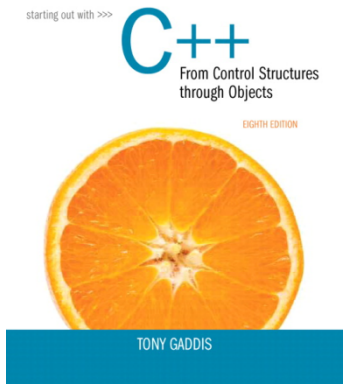
```
SomeClass::SomeClass
                    (const SomeClass &obj)
```

Contents of StudentTestScores.h (Version 2)

```
1  #ifndef STUDENTTESTSCORES_H
2  #define STUDENTTESTSCORES_H
3  #include <string>
4  using namespace std;
5
6  const double DEFAULT_SCORE = 0.0;
7
8  class StudentTestScores
9  {
10 private:
11    string studentName;  // The student's name
12    double *testScores;  // Points to array of test scores
13    int numTestScores;   // Number of test scores
14
15     // Private member function to create an
16     // array of test scores.
17    void createTestScoresArray(int size)
18    { numTestScores = size;
19      testScores = new double[size];
20      for (int i = 0; i < size; i++)
21         testScores[i] = DEFAULT_SCORE; }
22
23 public:
24     // Constructor
25    StudentTestScores(string name, int numScores)
26    { studentName = name;
```

```
27      createTestScoresArray(numScores); }
28
29    // Copy constructor
30    StudentTestScores(const StudentTestScores &obj)
31    { studentName = obj.studentName;
32      numTestScores = obj.numTestScores;
33      testScores = new double[numTestScores];
34      for (int i = 0; i < numTestScores; i++)
35         testScores[i] = obj.testScores[i]; }
36
37     // Destructor
38    ~StudentTestScores()
39    { delete [] testScores; }
40
41     // The setTestScore function sets a specific
42     // test score's value.
43    void setTestScore(double score, int index)
44    { testScores[index] = score; }
45
46     // Set the student's name.
47    void setStudentName(string name)
48    { studentName = name; }
49
50     // Get the student's name.
51    string getStudentName() const
52    { return studentName; }
```

```
53
54      // Get the number of test scores.
55      int getNumTestScores() const
56      { return numTestScores; }
57
58       // Get a specific test score.
59      double getTestScore(int index) const
60      { return testScores[index]; }
61  };
62  #endif
```



14.5

Operator Overloading

Addison-Wesley
is an imprint of
PEARSON

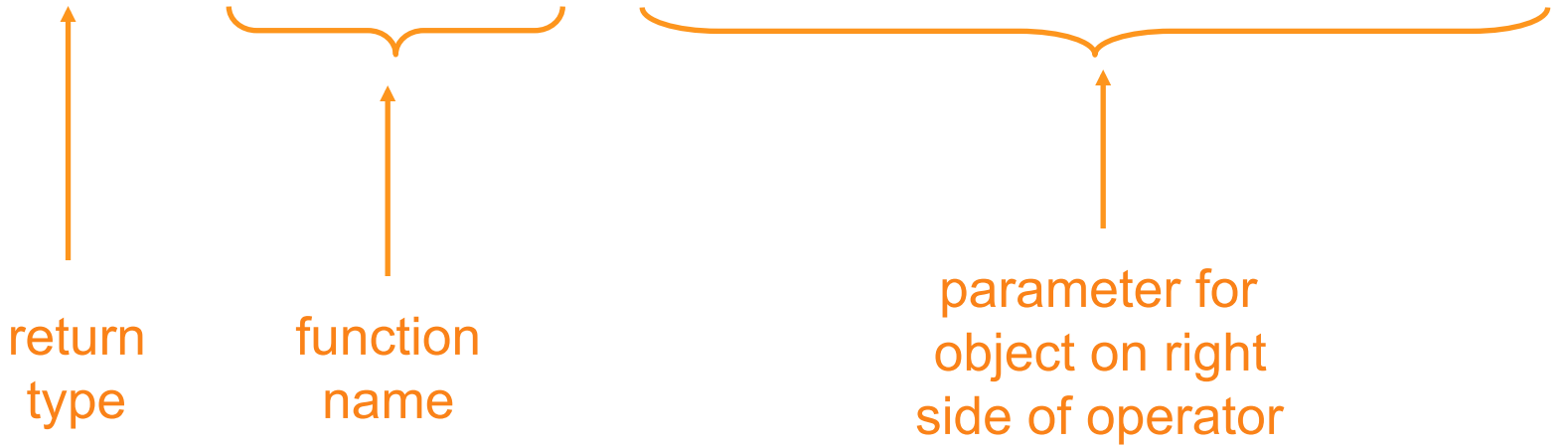
Operator Overloading

- 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

Operator Overloading

- Prototype:

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



- Operator is called via object on left side

Invoking an Overloaded Operator

- 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
{
  public:
    double operator-(const point2d &right)
    { return sqrt(pow((x-right.x),2)
                + pow((y-right.y),2)); }
...
  private:
    int x, y;
};
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;
```

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
- Can be used to access members that may be hidden by parameters with same name

`this` Pointer Example

```
class SomeClass
{
   private:
        int num;
   public:
        void setNum(int num)
        { this->num = num; }
        ...
};
```

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:

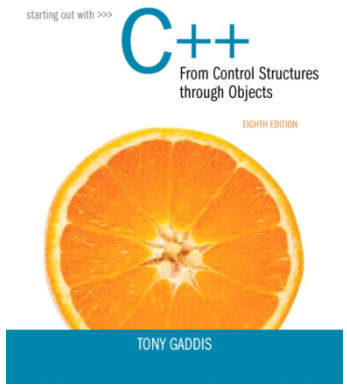
```
?:   .   .*   :: sizeof
```

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



14.6

Object Conversion

Addison-Wesley
is an imprint of
PEARSON

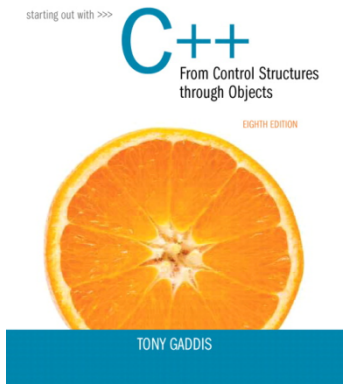
Object Conversion

- 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;
```



14.7

Aggregation

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;
  ...
};
```

See the `Instructor`, `TextBook`, and `Course` classes in Chapter 14.