

Structures and Classes

- In C++, the definition of a **structure** has been expanded so that it can also include member functions, including constructors and destructor functions, in just the same way that a class can.
- The only difference between a structure and a class is that, by default, the members of a class are **private** but the members of a structure are **public**.

Constructors & Destructors

- A constructor is called each time an object of that class is created.
- Any **initializations** that need to be performed on an object can be done automatically by the constructor function.
- A constructor function has the same name as the class of which it is a part and has **no return type**.

Constructors & Destructors (cont)

- **Data member** of a class **cannot** be initialized in the class definition.
- A destructor function is declared as preceding the class name with a **~**.

Example

```
class Time {
```

```
    // Time abstract data type (ADT) definition.
```

```
public:
```

```
    Time()    { hour = minute = second = 0; }
```

```
                // constructor
```

```
    ~Time();    // destructor
```

```
private:
```

```
    int hour, minute, second;
```

```
};
```

Constructor Taking Parameters

- *Time(int = 0 , int = 0, int = 0);*
// default arguments
- Destructor functions may **not** have parameters.

Inheritance

- All **public** elements of the base class will also public elements of the derived class.
- All **private** elements of the base elements remain private to it and are **not** directly accessible by the derived class.

Example

// Define base class

```
class Base {  
    int i;  
  
    public:  
        void set_i(int n);  
        int get_i();  
};
```

// Define derived class

```
class Derived : public Base {  
    int j;  
  
    public:  
        void set_j(int n);  
        int mul()          { return j * get_i(); }  
}
```

Example (cont)

Derived ob;

<i>ob.set_i(10);</i>	<i>// load i in Base</i>
<i>ob.set_j(4);</i>	<i>// load j in Derived</i>
<i>cout << ob.mul();</i>	<i>// display 40</i>

"const" Member Functions & Data Members

- A member function can be declared to be able to read but not write the object for which it is called.

E.g., ***int getHour () const {};*** // return hour

- A **non-const** member function **cannot** be called for a **const** object.

E.g.,

void setHour (int); // set hour

const Time noon (12, 0, 0); // constant object

noon.setHour (10); // illegal

"const" Member Functions & Data Members (2)

- It is better to declare "**const**" all member functions that do not need to modify the current object so that you can use them on a const object if you need to.
- The const declaration is **not** allowed for **constructors** and **destructors**.

"const" Member Functions & Data Members (3)

- **const** data members must be initialized using **member initializer**.

E.g.,

```
class Increment {  
    const int count;  
    const int dummy;  
};  
Increment (int c, int d) : count(c),  
                        dummy(d) {};
```

Objects as Members of Classes

- Objects are **constructed** from the **inside out** and **deconstructed** in the reverse order from the **outside in**.
- Better to initialize member objects explicitly through **member initializers**.

```
Employee (char fname, int bmonth, int hyear)  
    : birthDate(bmonth),  
      hireDate(hyear) {};
```

This eliminates the overhead of "**doubly initializing**" member objects, i.e., once when the member object's default **constructor** is called and again when **set** functions are used to initialize the member object.

"friend" Functions

- Not a member of a class but still has access to its **private** members.
- It is **not** possible to call a friend function by using an object name and a class member access operator (a dot or arrow). Instead, friends are called just like ***regular functions***.

"friend" Functions (cont)

- Will typically be passed one or more **objects** of the class for which they are defined to operate upon. E.g.,

```
class Count {  
    // friend declaration  
    friend void setX(Count &c, int val)    { c.x = val; }  
private:  
    int x;                // private data member  
};  
  
Count cnt;  
setX(cnt, 8);            // set x with a friend
```

“friend” Classes

```
class class_one {  
    friend class class_two;
```

- Friendship is granted, not taken.
- Friendship is neither symmetric nor transitive.

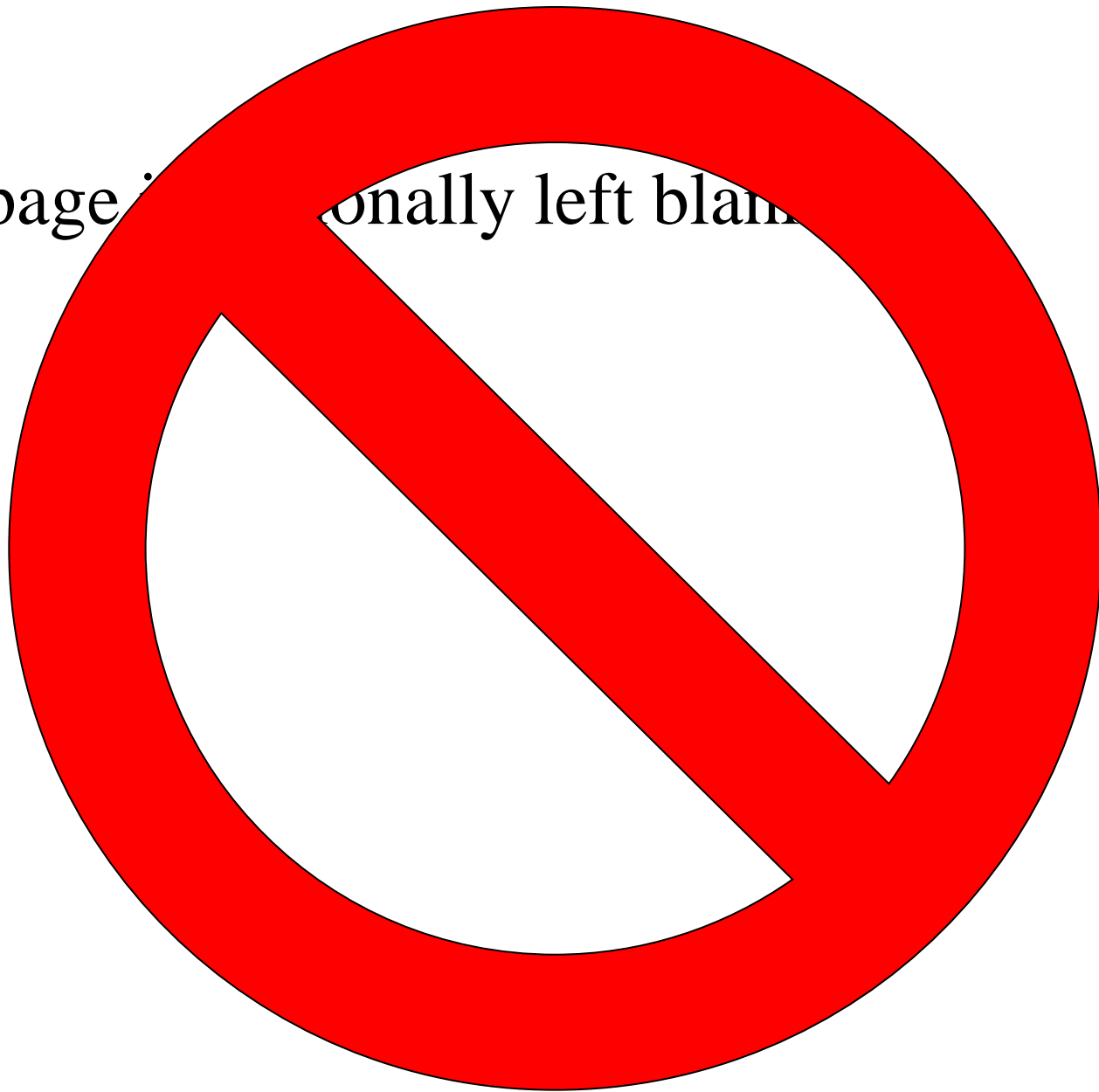
The "this" Pointer

- "this" is a pointer that is automatically passed to any member function when it is called.
- Only member functions are passed a pointer, therefore, a **friend** does not have "this" pointer.
- **(*this).data-member**, where parentheses are needed because the dot operator has higher precedence than the * operator.

Static Class Members

- Only **one** copy of the static member variable exists – no matter how many objects of that class are created.
- Static member variable exists **before** any object of its class is created.
- Must be called by prefixing its name with the **class** name and binary scope resolution operator.
- Within a static member function, there is **no** "**this**" pointer.

This page intentionally left blank



Chapter 6: Classes and Data Abstraction

Outline

- 6.1 Introduction
- 6.2 Structure Definitions
- 6.3 Accessing Structure Members
- 6.4 Implementing a User-Defined Type `Time` with a `struct`
- 6.5 Implementing a `Time` Abstract Data Type with a `class`
- 6.6 Class Scope and Accessing Class Members
- 6.7 Separating Interface from Implementation
- 6.8 Controlling Access to Members
- 6.9 Access Functions and Utility Functions
- 6.10 Initializing Class Objects: Constructors
- 6.11 Using Default Arguments with Constructors
- 6.12 Destructors
- 6.13 When Constructors and Destructors Are Called
- 6.14 Using *Set* and *Get* Functions
- 6.15 Subtle Trap: Returning a Reference to a `private` Data Member
- 6.16 Default Memberwise Assignment
- 6.17 Software Reusability



6.1 Introduction

- Object-oriented programming (OOP)
 - Encapsulates data (attributes) and functions (behavior) into packages called classes
- Information hiding
 - Class objects communicate across well-defined interfaces
 - Implementation details hidden within classes themselves
- User-defined (programmer-defined) types: classes
 - Data (data members)
 - Functions (member functions or methods)
 - Class instance: object



6.2 Structure Definitions

- Structures
 - Aggregate data types built using elements of other types

```
struct Time {  
    int hour;  
    int minute;  
    int second;  
};
```

Diagram illustrating the structure definition:

- The **Structure tag** points to the `struct Time` part of the definition.
- The **Structure members** point to the list of members: `int hour;`, `int minute;`, and `int second;`.

- Structure member naming
 - In same **struct**: must have unique names
 - In different **structs**: can share name
- **struct** definition must end with semicolon



6.4 Implementing a User-Defined Type `Time` with a `struct`

- Default: structures passed by value
 - Pass structure by reference
 - Avoid overhead of copying structure
- C-style structures
 - No “interface”
 - If implementation changes, all programs using that `struct` must change accordingly
 - Cannot `print` as unit
 - Must print/format member by member
 - Cannot `compare` in entirety
 - Must compare member by member





fig06_01.cpp
(3 of 3)

```

49 // print time in universal-time format
50 void printUniversal( const Time &t )
51 {
52     cout << setfill( '0' ) << setw( 2 ) << t.hour << ":"
53         << setw( 2 ) << t.minute << ":"
54         << setw( 2 ) << t.second;
55 } // end function printUniversal
56
57 // print time in standard-time format
58 void printStandard( const Time &t )
59 {
60     cout << ( ( t.hour == 0 || t.hour == 12 ) ?
61             12 : t.hour % 12 ) << ":" << setfill( '0' )
62             << setw( 2 ) << t.minute << ":"
63             << setw( 2 ) << t.second
64             << ( t.hour < 12 ? " AM" : " PM" );
65 } // end function printStandard

```

Use parameterized stream manipulator **setfill**.

Use dot operator to access data members.

Dinner will be held at 18:30:00 universal time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:00

6.5 Implementing a Time Abstract Data Type with a class

- Constructor function
 - Special member function
 - Initializes data members
 - Same name as class
 - Called when object instantiated
 - Several constructors
 - Function overloading
 - No return type



6.5 Implementing a Time Abstract Data Type with a class

- Member functions defined outside class
 - Binary scope resolution operator (`::`)
 - “Ties” member name to class name
 - Uniquely identify functions of particular class
 - Different classes can have member functions with same name
- Member functions defined **inside** class
 - Do not need scope resolution operator, class name
 - Compiler **attempts inline**
 - Outside class, inline explicitly with keyword **inline**



**fig06_03.cpp**
(1 of 5)

```
1  // Fig. 6.3: fig06_03.cpp
2  // Time class.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setfill;
11 using std::setw;
12
13 // Time abstract data type (ADT) definition
14 class Time {
15
16 public:
17     Time();                // constructor
18     void setTime( int, int, int ); // set hour, minute, second
19     void printUniversal();    // print universal-time format
20     void printStandard();    // print standard-time format
21
```

Define class **Time**.





fig06_03.cpp
(2 of 5)

```
22 private:
23     int hour;        // 0 - 23 (24-hour clock format)
24     int minute;      // 0 - 59
25     int second;      // 0 - 59
26
27 }; // end class Time
28
29 // Time constructor initializes each data member to 0
30 // ensures all Time objects start in a consistent state
31 Time::Time()
32 {
33     hour = minute = second = 0;
34
35 } // end Time constructor
36
37 // set new Time value using universal time, perform validity
38 // checks on the data values and set invalid values to zero
39 void Time::setTime( int h, int m, int s )
40 {
41     hour = ( h >= 0 && h < 24 ) ? h : 0;
42     minute = ( m >= 0 && m < 60 ) ? m : 0;
43     second = ( s >= 0 && s < 60 ) ? s : 0;
44
45 } // end function setTime
46
```

Constructor initializes
private data members
to 0.

public member
function checks
parameter values for
validity before setting
private data
members.



fig06_03.cpp
(3 of 5)

```
47 // print Time in universal format
48 void Time::printUniversal()
49 {
50     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
51         << setw( 2 ) << minute << ":"
52         << setw( 2 ) << second;
53
54 } // end function printUniversal
55
56 // print Time in standard format
57 void Time::printStandard()
58 {
59     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
60         << ":" << setfill( '0' ) << setw( 2 ) << minute
61         << ":" << setw( 2 ) << second
62         << ( hour < 12 ? " AM" : " PM" );
63
64 } // end function printStandard
65
66 int main()
67 {
68     Time t; // instantiate object t of class Time
69
```

No arguments (implicitly “know” purpose is to print data members); member function calls more concise.

Declare variable **t** to be object of class **Time**.

6.5 Implementing a Time Abstract Data Type with a class

- Destructors
 - Same name as class
 - Preceded with tilde (~)
 - No arguments
 - Cannot be overloaded
 - Performs “termination housekeeping”



6.6 Class Scope and Accessing Class Members

- Class scope
 - Data members, member functions
 - Within class scope
 - Class members
 - Immediately accessible by all member functions
 - Referenced by name
 - Outside class scope
 - Referenced through handles
 - Object name, reference to object, pointer to object



time1.h (1 of 1)

```

1  // Fig. 6.5: time1.h
2  // Declaration of class Time.
3  // Member functions are defined in t1.cpp
4
5  // prevent multiple inclusions
6  #ifndef TIME1_H
7  #define TIME1_H
8
9  // Time abstract class
10 class Time {
11
12 public:
13     Time(); // constructor
14     void setTime( int, int, int ); // set time in second,
15     void printUniversal(); // print time in universal time format
16     void printStandard(); // print time in standard time format
17
18 private:
19     int hour; // 0 - 23 (24-hour clock format)
20     int minute; // 0 - 59
21     int second; // 0 - 59
22
23 }; // end class Time
24
25 #endif

```

Preprocessor code to prevent multiple inclusions.

Code between these directives not included if name **TIME1_H** already defined.

“If not defined”

Preprocessor directive defines name **TIME1_H**.

Naming convention: header file name with underscore replacing period.

time1.cpp (1 of 3)

```
1  // Fig. 6.6: time1.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4
5  using std::cout;
6
7  #include <iomanip>
8
9  using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time1.h
13 #include "time1.h"
14
15 // Time constructor initializes each data member to zero.
16 // Ensures all Time objects
17 Time::Time()
18 {
19     hour = minute = second = 0;
20 }
21 // end Time constructor
22
```

Include header file
time1.h.

Name of header file enclosed
in quotes; angle **brackets**
cause preprocessor to assume
header part of C++ Standard
Library.

6.8 Controlling Access to Members

- Class member access
 - Default **private**
 - Explicitly set to **private**, **public**, **protected**
- **struct** member access
 - Default **public**
 - Explicitly set to **private**, **public**, **protected**
- Access to class's **private** data
 - Controlled with access functions (accessor methods)
 - Get function
 - Read **private** data
 - Set function
 - Modify **private** data



6.9 Access Functions and Utility Functions

- Access functions
 - **public**
 - Read/display data
 - Predicate functions
 - Check conditions
- Utility functions (helper functions)
 - **private**
 - Support operation of **public** member functions
 - Not intended for direct client use



6.11 Using Default Arguments with Constructors

- Constructors
 - Can specify default arguments
 - Default constructors
 - Defaults all arguments
- OR
- Explicitly requires no arguments
 - Can be invoked with no arguments
 - Only one per class



6.12 Destructors

- Destructors
 - Special member function
 - Same name as class
 - Preceded with tilde (~)
 - No arguments
 - No return value
 - Cannot be overloaded
 - Performs “termination housekeeping”
 - Before system reclaims object’s memory
 - Reuse memory for new objects
 - No explicit destructor
 - Compiler creates “empty” destructor”



6.13 When Constructors and Destructors Are Called

- Order of constructor, destructor function calls
 - **Global** scope objects
 - Constructors
 - Before any other function (including **main**)
 - Destructors
 - When **main** terminates (or **exit** function called)
 - Not called if program terminates with **abort**
 - **Automatic** local objects
 - Constructors
 - When objects defined
 - Each time execution enters scope
 - Destructors
 - When objects leave scope
 - Execution exits block in which object defined
 - Not called if program ends with **exit** or **abort**



6.13 When Constructors and Destructors Are Called

- Order of constructor, destructor function calls
 - **static** local objects
 - Constructors
 - Exactly **once**
 - When execution reaches point where object defined
 - Destructors
 - When **main** terminates or **exit** function called
 - Not called if program ends with **abort**




6.15 Subtle Trap: Returning a Reference to a `private` Data Member

- Returning references
 - **public** member functions can return non-**const** references to **private** data members
 - Client able to modify **private** data members



**time4.h (1 of 1)**

```
1  // Fig. 6.21: time4.h
2  // Declaration of class Time.
3  // Member functions defined in time4.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME4_H
7  #define TIME4_H
8
9  class Time {
10
11  public:
12      Time( int = 0, int = 0, int = 0 );
13      void setTime( int, int, int );
14      int getHour();
15
16      int &badSetHour( int ); // DANGEROUS reference return
17
18  private:
19      int hour;
20      int minute;
21      int second;
22
23  }; // end class Time
24
25  #endif
```



Function to demonstrate
effects of returning reference
to **private** data member.



```
25 // return hour value
26 int Time::getHour()
27 {
28     return hour;
29 }
30 // end function getHour
31
32 // POOR PROGRAMMING PRACTICE:
33 // Returning a reference to a private data member.
34 int &Time::badSetHour( int hh )
35 {
36     hour = ( hh >= 0 && hh < 24 )
37
38     return hour; // DANGEROUS reference return
39
40 } // end function badSetHour
```

Return reference to **private**
data member **hour**.

6.16 Default Memberwise Assignment

- Assigning objects
 - Assignment operator (=)
 - Can assign one object to another of same type
 - Default: **memberwise** assignment
 - Each right member assigned individually to left member



This page is intentionally left blank



Chapter 7: Classes Part II

Outline

- 7.1 Introduction
- 7.2 `const` (Constant) Objects and `const` Member Functions
- 7.3 Composition: Objects as Members of Classes
- 7.4 friend Functions and friend Classes
- 7.5 Using the `this` Pointer
- 7.6 Dynamic Memory Allocation with Operators `new` and `delete`
- 7.7 `static` Class Members
- 7.8 Data Abstraction and Information Hiding
 - 7.8.1 Example: Array Abstract Data Type
 - 7.8.2 Example: String Abstract Data Type
 - 7.8.3 Example: Queue Abstract Data Type
- 7.9 Container Classes and Iterators
- 7.10 Proxy Classes



7.2 **const** (Constant) Objects and **const** Member Functions

- Principle of least privilege
 - Only give objects permissions they need, no more
- Keyword **const**
 - Specify that an object is not modifiable
 - Any attempt to modify the object is a syntax error
 - Example
 - ```
const Time noon(12, 0, 0);
```
    - Declares a **const** object **noon** of class **Time** and initializes it to 12



## 7.2 **const** (Constant) Objects and **const** Member Functions

- **const** objects require **const** functions
  - Member functions declared **const** **cannot** modify their object
  - **const** must be specified in function prototype and definition
  - Prototype:  
*ReturnType FunctionName(param1,param2...) **const**;*
  - Definition:  
*ReturnType FunctionName(param1,param2...) **const** { ... }*
  - Example:  

```
int A::getValue() const { return
 privateDataMember };
```

    - Returns the value of a data member but doesn't modify anything so is declared **const**
- Constructors / Destructors **cannot** be **const**
  - They need to initialize variables, therefore modifying them



## 7.2 **const (Constant) Objects and const Member Functions**

- Member initializer syntax
  - Data member increment in class **Increment**
  - constructor for **Increment** is modified as follows:

```
Increment::Increment(int c, int i)
 : increment(i)
 { count = c; }
```
  - `: increment( i )` initializes increment to **i**
  - All data members **can** be initialized using member initializer syntax
  - **consts** and references **must** be initialized using member initializer syntax
  - Multiple member initializers
    - Use comma-separated list after the colon



## 7.3 Composition: Objects as Members of Classes

- Composition
  - Class has objects of other classes as members
- Construction of objects
  - Member objects constructed in **order** declared
    - **Not** in order of constructor's member initializer list
  - Constructed before their enclosing class objects (host objects)





## 7.4 friend Functions and friend Classes

- **friend** function and **friend** classes
  - Can access **private** and **protected** members of another class
  - **friend** functions are **not** member functions of class
    - Defined outside of class scope
- Properties of friendship
  - Friendship is granted, not taken
  - Not symmetric (if **B** a **friend** of **A**, **A** not necessarily a **friend** of **B**)
  - Not transitive (if **A** a **friend** of **B**, **B** a **friend** of **C**, **A** not necessarily a **friend** of **C**)



## 7.4 friend Functions and friend Classes

- **friend** declarations

- To declare a **friend** function

- Type **friend** before the function prototype in the class that is giving friendship

- `friend int myFunction( int x );`

- should appear in the class giving friendship

- To declare a **friend** class

- Type **friend class Classname** in the class that is giving friendship

- if **ClassOne** is granting friendship to **ClassTwo**,

- `friend class ClassTwo;`

- should appear in **ClassOne**'s definition



## 1. Class definition

### 1.1 Declare function a friend

### 1.2 Function definition

### 1.3 Initialize Count object

**setX** a friend of class **Count** (can access private data).

**setX** is defined normally and is not a member function of **Count**.

Changing **private** variables allowed.

```

1 // Fig. 7.5: fig07_05.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // Modified Count class
9 class Count {
10 friend void setX(Count &, int); // friend declaration
11 public:
12 Count() { x = 0; } // constructor
13 void print() const { cout << x << endl; } // output
14 private:
15 int x; // data member
16 };
17
18 // Can modify private data of Count because
19 // setX is declared as a friend function of
20 void setX(Count &c, int val)
21 {
22 c.x = val; // legal: setX is a friend of Count
23 }
24
25 int main()
26 {
27 Count counter;
28
29 cout << "counter.x after instantiation: ";
30 counter.print();

```

## 7.5 Using the `this` Pointer

- **this** pointer
  - Allows objects to access their own address
  - Not part of the object itself
  - Implicit **first** argument on non-static member function call to the object
  - Implicitly reference member data and functions
  - The type of the **this** pointer depends upon the type of the object and whether the member function using **this** is **const**
  - In a non-**const** member function of **Employee**, **this** has type  
**Employee \* const**
    - **Constant** pointer to an **Employee** object
  - In a **const** member function of **Employee**, **this** has type  
**const Employee \* const**
    - Constant pointer to a constant **Employee** object



## 7.5 Using the **this** Pointer

- Examples using **this**
  - For a member function print data member **x**, either  
`this->x`  
or  
`( *this ).x`
- Cascaded member function calls
  - Function returns a reference pointer to the same object  
`{ return *this; }`
  - Other functions can operate on that pointer
  - Functions that do not return references must be called last



## 7.5 Using the this Pointer

- Example of cascaded member function calls
  - Member functions **setHour**, **setMinute**, and **setSecond** all return **\*this** (reference to an object)
  - For object **t**, consider  
`t.setHour(1).setMinute(2).setSecond(3);`
  - Executes **t.setHour(1)**, returns **\*this** (reference to object) and the expression becomes  
`t.setMinute(2).setSecond(3);`
  - Executes **t.setMinute(2)**, returns reference and becomes  
`t.setSecond(3);`
  - Executes **t.setSecond(3)**, returns reference and becomes  
`t;`
  - Has no effect



## 7.7 static Class Members

- **static** class members
  - Shared by all objects of a class
    - Normally, each object gets its own copy of each variable
  - Efficient when a single copy of data is enough
    - Only the **static** variable has to be updated
  - May seem like global variables, but have class scope
    - only accessible to objects of same class
  - Initialized at **file** scope
  - Exist even if **no** instances (objects) of the class exist
  - Both variables and functions can be **static**



## 7.7 static Class Members

- **static** variables

- Static variables are accessible through any object of the class
- **public static** variables
  - Can also be accessed using scope resolution operator( :: )

**Employee::count**

- **private static** variables

- When no class member objects exist, can only be accessed via a **public static** member function
  - To call a **public static** member function combine the class name, the :: operator and the function name

**Employee::getCount( )**





## 7.7 static Class Members

- **Static** functions
  - **static** member functions **cannot** access non-**static** data or functions
  - There is no **this** pointer for **static** functions, they exist independent of objects

