

A decorative vertical bar on the left side of the slide. It features a dark teal background with a white dotted vertical line. To the right of this bar are several orange circles of varying sizes, arranged in a cluster. The main title text is positioned to the right of this decorative area.

OBJECT ORIENTED PROGRAMMING USING C++



Outline

- 17.1 Introduction
- 17.2 `const` (Constant) Objects and `const` Member Functions
- 17.3 Composition: Objects as Members of Classes
- 17.4 `friend` Functions and `friend` Classes
- 17.5 Using the `this` Pointer
- 17.6 Dynamic Memory Allocation with Operators `new` and `delete`
- 17.7 `static` Class Members
- 17.8 Data Abstraction and Information Hiding
 - 17.8.1 Example: Array Abstract Data Type
 - 17.8.2 Example: String Abstract Data Type
 - 17.8.3 Example: Queue Abstract Data Type
- 17.9 Container Classes and Iterators



17.1 Introduction

- Chapters 16-18
 - Object-based programming
- Chapter 19-20
 - Polymorphism and inheritance
 - Object-oriented programming



17.2 `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
 - For example:

```
const time noon( 12, 0, 0 );
```
 - Declares a **`const`** object **`noon`** of class **`time`** and initializes it to 12 noon



17.2 **const** (Constant) Objects and **const** Member Functions (II)

- **const** objects require **const** functions

- Functions declared **const** cannot modify the object
- **const** 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, and is appropriately declared **const**
- Constructors / Destructors cannot be **const**
 - They need to initialize variables (therefore modifying them)



```

1 // Fig. 17.1: time5.h
2 // Declaration of the class Time.
3 // Member functions defined in time5.cpp
4 #ifndef TIME5_H
5 #define TIME5_H
6
7 class Time {
8 public:
9     Time( int = 0, int = 0, int = 0 ); // default constructor
10
11     // set functions
12     void setTime( int, int, int ); // set time
13     void setHour( int ); // set hour
14     void setMinute( int ); // set minute
15     void setSecond( int ); // set second
16
17     // get functions (normally declared const)
18     int getHour() const; // return hour
19     int getMinute() const; // return minute
20     int getSecond() const; // return second
21
22     // print functions (normally declared const)
23     void printMilitary() const; // print military time
24     void printStandard(); // print standard time
25 private:
26     int hour; // 0 - 23
27     int minute; // 0 - 59
28     int second; // 0 - 59
29 };
30
31 #endif

```



Outline

1. Class definition

1.1 Function prototypes

1.2 Member variables

```
32 // Fig. 17.1: time5.cpp
33 // Member function definitions for Time class.
34 #include <iostream>
35
36 using std::cout;
37
38 #include "time5.h"
39
40 // Constructor function to initialize private data.
41 // Default values are 0 (see class definition).
42 Time::Time( int hr, int min, int sec )
43     { setTime( hr, min, sec ); }
44
45 // Set the values of hour, minute, and second.
46 void Time::setTime( int h, int m, int s )
47 {
48     setHour( h );
49     setMinute( m );
50     setSecond( s );
51 }
52
53 // Set the hour value
54 void Time::setHour( int h )
55     { hour = ( h >= 0 && h < 24 ) ? h : 0; }
56
57 // Set the minute value
58 void Time::setMinute( int m )
59     { minute = ( m >= 0 && m < 60 ) ? m : 0; }
60
61 // Set the second value
62 void Time::setSecond( int s )
63     { second = ( s >= 0 && s < 60 ) ? s : 0; }
```



Outline



Source Code

1. Load Header

1.1 Function definitions

```

64
65 // Get the hour value
66 int Time::getHour() const { return hour; }
67
68 // Get the minute value
69 int Time::getMinute() const { return minute; }
70
71 // Get the second value
72 int Time::getSecond() const { return second; }
73
74 // Display military format time: HH:MM
75 void Time::printMilitary() const
76 {
77     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
78         << ( minute < 10 ? "0" : "" ) << minute;
79 }
80
81 // Display standard format time: HH:MM:SS AM (or PM)
82 void Time::printStandard() // should be const
83 {
84     cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
85         << ( minute < 10 ? "0" : "" ) << minute << ":"
86         << ( second < 10 ? "0" : "" ) << second
87         << ( hour < 12 ? " AM" : " PM" );
88 }

```



Outline

1.1 Function definitions

1.2 Purposely leave out const keyword for printStandard


```

89 // Fig. 17.1: fig17_01.cpp
90 // Attempting to access a const object with
91 // non-const member functions.
92 #include "time5.h"
93
94 int main()
95 {
96     Time wakeUp( 6, 45, 0 );           // non-constant object
97     const Time noon( 12, 0, 0 );      // constant object
98
99                                     // MEMBER FUNCTION   OBJECT
100    wakeUp.setHour( 18 );              // non-const       non-const
101
102    noon.setHour( 12 );                // non-const       const
103
104    wakeUp.getHour();                  // const           non-const
105
106    noon.getMinute();                  // const           const
107    noon.printMilitary();              // const           const
108    noon.printStandard();              // non-const       const
109    return 0;
110 }

```



Outline



1. Initialize variables
2. Attempt to use non-const functions with const objects

```

Compiling...
Fig07_01.cpp
d:fig07_01.cpp(14) : error C2662: 'setHour' : cannot convert 'this'
pointer from 'const class Time' to 'class Time &'
Conversion loses qualifiers
d:\fig07_01.cpp(20) : error C2662: 'printStandard' : cannot convert
'this' pointer from 'const class Time' to 'class Time &'
Conversion loses qualifiers
Time5.cpp
Error executing cl.exe.

test.exe - 2 error(s), 0 warning(s)

```

Program Output

17.2 **const (Constant) Objects and const Member Functions (III)**

- 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 the value of **i**.
 - Any data member can be initialized using member initializer syntax
 - **const**s and references must be initialized this way
- Multiple member initializers
 - Use comma-separated list after the colon



```

1 // Fig. 17.2: fig17_02.cpp
2 // Using a member initializer to initialize a
3 // constant of a built-in data type.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 class Increment {
10 public:
11     Increment( int c = 0, int i = 1 );
12     void addIncrement() { count += increment; }
13     void print() const;
14
15 private:
16     int count;
17     const int increment; // const data member
18 };
19
20 // Constructor for class Increment
21 Increment::Increment( int c, int i )
22     : increment( i ) // initializer for const member
23 { count = c; }
24
25 // Print the data
26 void Increment::print() const
27 {
28     cout << "count = " << count
29         << ", increment = " << increment << endl;
30 }
31
32 int main()
33 {

```



Outline

1. Class definition

1.1 Function definitions

```
34  Increment value( 10, 5 );
35
36  cout << "Before incrementing: ";
37  value.print();
38
39  for ( int j = 0; j < 3; j++ ) {
40      value.addIncrement();
41      cout << "After increment " << j + 1 << ": ";
42      value.print();
43  }
44
45  return 0;
46 }
```



Outline

1.2 Initialize variables

2. Function calls

3. Output results

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

17.3 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)
 - Constructors called inside out
 - Destructors called outside in



17.3 Composition: Objects as Members of Classes (II)

- Example:

```
Employee::Employee( char *fname, char *lname,  
                    int bmonth, int bday, int byear,  
                    int hmonth, int hday, int hyear )  
: birthDate( bmonth, bday, byear ),  
  hireDate( hmonth, hday, hyear )
```

- Insert objects from **Date** class (**birthDate** and **hireDate**) into **Employee** class
- **birthDate** and **hireDate** have member initializers - they are probably **consts** in the **Employee** class



```
1 // Fig. 17.4: date1.h
2 // Declaration of the Date class.
3 // Member functions defined in date1.cpp
4 #ifndef DATE1_H
5 #define DATE1_H
6
7 class Date {
8 public:
9     Date( int = 1, int = 1, int = 1900 ); // default constructor
10    void print() const; // print date in month/day/year format
11    ~Date(); // provided to confirm destruction order
12 private:
13    int month; // 1-12
14    int day; // 1-31 based on month
15    int year; // any year
16
17    // utility function to test proper day for month and year
18    int checkDay( int );
19 };
20
21 #endif
```



Outline



1. Class definition

1.1 Member functions

1.2 Member variables

```

22 // Fig. 17.4: date1.cpp
23 // Member function definitions for Date class.
24 #include <iostream>
25
26 using std::cout;
27 using std::endl;
28
29 #include "date1.h"
30
31 // Constructor: Confirm proper value for month;
32 // call utility function checkDay to confirm proper
33 // value for day.
34 Date::Date( int mn, int dy, int yr )
35 {
36     if ( mn > 0 && mn <= 12 )           // validate the month
37         month = mn;
38     else {
39         month = 1;
40         cout << "Month " << mn << " invalid. Set to month 1.\n";
41     }
42
43     year = yr;                          // should validate yr
44     day = checkDay( dy );                // validate the day
45
46     cout << "Date object constructor for date ";
47     print();                             // interesting: a print with no arguments
48     cout << endl;
49 }
50

```



Outline



1. Load header

1.1 Function definitions

1.2 Date constructor


```

51 // Print Date object in form month/day/year
52 void Date::print() const
53     { cout << month << '/' << day << '/' << year; }
54
55 // Destructor: provided to confirm destruction order
56 Date::~Date()
57 {
58     cout << "Date object destructor for date ";
59     print();
60     cout << endl;
61 }
62
63 // Utility function to confirm proper day value
64 // based on month and year.
65 // Is the year 2000 a leap year?
66 int Date::checkDay( int testDay )
67 {
68     static const int daysPerMonth[ 13 ] =
69         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
70
71     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
72         return testDay;
73
74     if ( month == 2 &&          // February: Check for leap year
75         testDay == 29 &&
76         ( year % 400 == 0 ||
77         ( year % 4 == 0 && year % 100 != 0 ) ) )
78         return testDay;
79
80     cout << "Day " << testDay << " invalid. Set to day 1.\n";
81
82     return 1; // leave object in consistent state if bad value
83 }

```



Outline



1.3 print function

1.4 Date destructor

1.5 checkDay function

```
84 // Fig. 17.4: emply1.h
85 // Declaration of the Employee class.
86 // Member functions defined in emply1.cpp
87 #ifndef EMPLOY1_H
88 #define EMPLOY1_H
89
90 #include "date1.h"
91
92 class Employee {
93 public:
94     Employee( char *, char *, int, int, int, int, int, int );
95     void print() const;
96     ~Employee(); // provided to confirm destruction order
97 private:
98     char firstName[ 25 ];
99     char lastName[ 25 ];
100    const Date birthDate;
101    const Date hireDate;
102};
103
104#endif
```



Outline



1. Load header

1.1 Class definition

1.2 Member functions

1.3 Member variables

1.3.1 Include const variables from Date class

```

105// Fig. 17.4: emply1.cpp
106// Member function definitions for Employee class.
107#include <iostream>
108
109using std::cout;
110using std::endl;
111
112#include <cstring>
113#include "emply1.h"
114#include "date1.h"
115
116Employee::Employee( char *fname, char *lname,
117                  int bmonth, int bday, int byear,
118                  int hmonth, int hday, int hyear )
119    : birthDate( bmonth, bday, byear ),
120      hireDate( hmonth, hday, hyear )
121{
122    // copy fname into firstName and be sure that it fits
123    int length = strlen( fname );
124    length = ( length < 25 ? length : 24 );
125    strncpy( firstName, fname, length );
126    firstName[ length ] = '\0';
127
128    // copy lname into lastName and be sure that it fits
129    length = strlen( lname );
130    length = ( length < 25 ? length : 24 );
131    strncpy( lastName, lname, length );
132    lastName[ length ] = '\0';
133
134    cout << "Employee object constructor: "
135          << firstName << ' ' << lastName << endl;
136}

```



Outline



1. Load header files

1.1 Function definitions

1.2 Employee constructor

1.2.1 Use member-initializer syntax for const Date members

```
137
138 void Employee::print() const
139 {
140     cout << lastName << ", " << firstName << "\nHired: ";
141     hireDate.print();
142     cout << " Birth date: ";
143     birthDate.print();
144     cout << endl;
145 }
146
147 // Destructor: provided to confirm destruction order
148 Employee::~Employee()
149 {
150     cout << "Employee object destructor: "
151           << lastName << ", " << firstName << endl;
152 }
```



Outline



1.3 print definition

**1.4 Employee
destructor**

```
153// Fig. 17.4: fig17_04.cpp
154// Demonstrating composition: an object with member objects.
155#include <iostream>
156
157using std::cout;
158using std::endl;
159
160#include "emply1.h"
161
162int main()
163{
164    Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );
165
166    cout << '\n';
167    e.print();
168
169    cout << "\nTest Date constructor with invalid values:\n";
170    Date d( 14, 35, 1994 ); // invalid Date values
171    cout << endl;
172    return 0;
173}
```



Outline

1. Load header files

2. Create Employee object

2.1 Attempt invalid Date setting

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones
```

```
Jones, Bob
Hired: 3/12/1988 Birth date: 7/24/1949
```

```
Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994
```

```
Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```



Outline

Program Output

17.4 friend Functions and friend Classes

- **friend** function and **friend** classes
 - Can access **private** and **protected** (more later) members of another class
 - **friend** functions are not member functions of class
 - Defined outside of class scope
- Properties
 - 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)



17.4 friend Functions and friend Classes (II)

- **friend** declarations

- **friend** function

- Keyword **friend** before function prototype in class that is giving friendship.

- **friend int myFunction(int x);**

- Appears in the class granting friendship

- **friend** class

- Type **friend class** *Classname* in class granting friendship
- If **ClassOne** granting friendship to **ClassTwo**,

friend class ClassTwo;

appears in **ClassOne**'s definition




```

1 // Fig. 17.5: fig17_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 Count
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();

```



Outline



1. Class definition

1.1 Declare function a friend

1.2 Function definition

1.3 Initialize Count object

```
31     cout << "counter.x after call to setX friend function: ";  
32     setX( counter, 8 ); // set x with a friend  
33     counter.print();  
34     return 0;  
35 }
```



Outline



2. Modify object

3. Print results

```
counter.x after instantiation: 0  
counter.x after call to setX friend function: 8
```

Program Output

```

1 // Fig. 17.6: fig17_06.cpp
2 // Non-friend/non-member functions cannot access
3 // private data of a class.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // Modified Count class
10 class Count {
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 // Function tries to modify private data of Count,
19 // but cannot because it is not a friend of Count.
20 void cannotSetX( Count &c, int val )
21 {
22     c.x = val; // ERROR: 'Count::x' is not accessible
23 }
24
25 int main()
26 {
27     Count counter;
28
29     cannotSetX( counter, 3 ); // cannotSetX is not a friend
30     return 0;
31 }

```



Outline

**(Previous program
without friend
declared)**

```
Compiling...
Fig07_06.cpp
D:\books\2000\cpphttp3\examples\Ch07\Fig07_06\Fig07_06.cpp(22) :
    error C2248: 'x' : cannot access private member declared in
    class 'Count'
        D:\books\2000\cpphttp3\examples\Ch07\Fig07_06\
        Fig07_06.cpp(15) : see declaration of 'x'
Error executing cl.exe.

test.exe - 1 error(s), 0 warning(s)
```



Outline

Program Output

17.5 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
- Example: class **Employee**
 - For non-**const** member functions: type **Employee * const**
 - Constant pointer to an **Employee** object
 - For **const** member functions: type **const Employee * const**
 - Constant pointer to an constant **Employee** object



17.5 Using the this Pointer (II)

- 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



17.5 Using the this Pointer (III)

- Example

- 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)` and returns `*this` (reference to object), and 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



```

1 // Fig. 17.7: fig17_07.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class Test {
9 public:
10     Test( int = 0 );           // default constructor
11     void print() const;
12 private:
13     int x;
14 };
15
16 Test::Test( int a ) { x = a; } // constructor
17
18 void Test::print() const // ( ) around *this required
19 {
20     cout << "          x = " << x
21          << "\n  this->x = " << this->x
22          << "\n(*this).x = " << ( *this ).x << endl;
23 }
24
25 int main()
26 {
27     Test testObject( 12 );
28
29     testObject.print();
30
31     return 0;
32 }

```



Outline



1. Class definition

1.1 Function definition

1.2 Initialize object

2. Function call


```
x = 12  
this->x = 12  
(*this).x = 12
```



Outline



Program Output

```

1 // Fig. 17.8: time6.h
2 // Cascading member function calls.
3
4 // Declaration of class Time.
5 // Member functions defined in time6.cpp
6 #ifndef TIME6_H
7 #define TIME6_H
8
9 class Time {
10 public:
11     Time( int = 0, int = 0, int = 0 ); // default constructor
12
13     // set functions
14     Time &setTime( int, int, int ); // set hour, minute, second
15     Time &setHour( int ); // set hour
16     Time &setMinute( int ); // set minute
17     Time &setSecond( int ); // set second
18
19     // get functions (normally declared const)
20     int getHour() const; // return hour
21     int getMinute() const; // return minute
22     int getSecond() const; // return second
23
24     // print functions (normally declared const)
25     void printMilitary() const; // print military time
26     void printStandard() const; // print standard time
27 private:
28     int hour; // 0 - 23
29     int minute; // 0 - 59
30     int second; // 0 - 59
31 };
32
33 #endif

```



Outline



Cascading function calls

1. Class definition

```

34 // Fig. 7.8: time.cpp
35 // Member function definitions for Time class.
36 #include <iostream>
37
38 using std::cout;
39
40 #include "time6.h"
41
42 // Constructor function to initialize private data.
43 // Calls member function setTime to set variables.
44 // Default values are 0 (see class definition).
45 Time::Time( int hr, int min, int sec )
46     { setTime( hr, min, sec ); }
47
48 // Set the values of hour, minute, and second.
49 Time &Time::setTime( int h, int m, int s )
50 {
51     setHour( h );
52     setMinute( m );
53     setSecond( s );
54     return *this;    // enables cascading
55 }
56
57 // Set the hour value
58 Time &Time::setHour( int h )
59 {
60     hour = ( h >= 0 && h < 24 ) ? h : 0;
61
62     return *this;    // enables cascading
63 }
64

```



Outline

1. Load header file

1.1 Function definitions

```

65 // Set the minute value
66 Time &Time::setMinute( int m )
67 {
68     minute = ( m >= 0 && m < 60 ) ? m : 0;
69
70     return *this;    // enables cascading
71 }
72
73 // Set the second value
74 Time &Time::setSecond( int s )
75 {
76     second = ( s >= 0 && s < 60 ) ? s : 0;
77
78     return *this;    // enables cascading
79 }
80
81 // Get the hour value
82 int Time::getHour() const { return hour; }
83
84 // Get the minute value
85 int Time::getMinute() const { return minute; }
86
87 // Get the second value
88 int Time::getSecond() const { return second; }
89
90 // Display military format time: HH:MM
91 void Time::printMilitary() const
92 {
93     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
94         << ( minute < 10 ? "0" : "" ) << minute;

```



Outline

1.1 Function definitions

```

95 }
96
97 // Display standard format time: HH:MM:SS AM (or PM)
98 void Time::printStandard() const
99 {
100     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
101         << ":" << ( minute < 10 ? "0" : "" ) << minute
102         << ":" << ( second < 10 ? "0" : "" ) << second
103         << ( hour < 12 ? " AM" : " PM" );
104 }
105 // Fig. 17.8: fig17_08.cpp
106 // Cascading member function calls together
107 // with the this pointer
108 #include <iostream>
109
110 using std::cout;
111 using std::endl;
112
113 #include "time6.h"
114
115 int main()
116 {
117     Time t;
118
119     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
120     cout << "Military time: ";
121     t.printMilitary();
122     cout << "\nStandard time: ";
123     t.printStandard();
124
125     cout << "\n\nNew standard time: ";
126     t.setTime( 20, 20, 20 ).printStandard();

```



Outline



1.1 Function definitions

Source File

1. Load header

1.1 Initialize Time object

2. Function calls

3. Print values

```
127 cout << endl;
```

```
128
```

```
129 return 0;
```

```
130 }
```



Outline

```
Military time: 18:30
```

```
Standard time: 6:30:22 PM
```

```
New standard time: 8:20:20 PM
```

Program Output

17.6 Dynamic Memory Allocation with Operators `new` and `delete`

- **new** and **delete**
 - Better dynamic memory allocation than C's **malloc** and **free**
 - **new** - automatically creates object of proper size, calls constructor, returns pointer of the correct type
 - **delete** - destroys object and frees space
- Example:
 - `TypeName *typeNamePtr;`
 - Creates pointer to a `TypeName` object
 - `typeNamePtr = new TypeName;`
 - **new** creates `TypeName` object, returns pointer (which `typeNamePtr` is set equal to)
 - `delete typeNamePtr;`
 - Calls destructor for `TypeName` object and frees memory



17.6 Dynamic Memory Allocation with Operators `new` and `delete` (II)

- Initializing objects

```
double *thingPtr = new double( 3.14159 );
```

- Initializes object of type `double` to `3.14159`

```
int *arrayPtr = new int[ 10 ];
```

- Create ten element `int` array, assign to `arrayPtr`.

- Use

```
delete [] arrayPtr;
```

to `delete` arrays



17.7 *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
- Can be variables or functions
 - **public**, **private**, or **protected**



17.7 static Class Members (II)

- Accessing **static** members
 - **public static** variables: accessible through any object of the class
 - Or use class name and (::)
Employee::count
 - **private static** variables: a **public static** member function must be used.
 - Prefix with class name and (::)
Employee::getCount()
 - **static** member functions cannot access non-**static** data or functions
 - No **this** pointer, function exists independent of objects



```
1 // Fig. 17.9: employ1.h
2 // An employee class
3 #ifndef EMPLOY1_H
4 #define EMPLOY1_H
5
6 class Employee {
7 public:
8     Employee( const char*, const char* ); // constructor
9     ~Employee(); // destructor
10    const char *getFirstName() const; // return first name
11    const char *getLastName() const; // return last name
12
13    // static member function
14    static int getCount(); // return # objects instantiated
15
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data member
21    static int count; // number of objects instantiated
22 };
23
24 #endif
```



Outline

1. Class definition

1.1 Function prototypes

1.2 Declare variables

```

25 // Fig. 17.9: employ1.cpp
26 // Member function definitions for class Employee
27 #include <iostream>
28
29 using std::cout;
30 using std::endl;
31
32 #include <cstring>
33 #include <cassert>
34 #include "employ1.h"
35
36 // Initialize the static data member
37 int Employee::count = 0;
38
39 // Define the static member function that
40 // returns the number of employee objects instantiated.
41 int Employee::getCount() { return count; }
42
43 // Constructor dynamically allocates space for the
44 // first and last name and uses strcpy to copy
45 // the first and last names into the object
46 Employee::Employee( const char *first, const char *last )
47 {
48     firstName = new char[ strlen( first ) + 1 ];
49     assert( firstName != 0 ); // ensure memory allocated
50     strcpy( firstName, first );
51
52     lastName = new char[ strlen( last ) + 1 ];
53     assert( lastName != 0 ); // ensure memory allocated
54     strcpy( lastName, last );
55
56     ++count; // increment static count of employees

```



Outline



1. Load header file

1.1 Initialize static data members

1.2 Function definitions

```

57     cout << "Employee constructor for " << firstName
58         << ' ' << lastName << " called." << endl;
59 }
60
61 // Destructor deallocates dynamically allocated memory
62 Employee::~Employee()
63 {
64     cout << "~Employee() called for " << firstName
65         << ' ' << lastName << endl;
66     delete [] firstName; // recapture memory
67     delete [] lastName;  // recapture memory
68     --count; // decrement static count of employees
69 }
70
71 // Return first name of employee
72 const char *Employee::getFirstName() const
73 {
74     // Const before return type prevents client from modifying
75     // private data. Client should copy returned string before
76     // destructor deletes storage to prevent undefined pointer.
77     return firstName;
78 }
79
80 // Return last name of employee
81 const char *Employee::getLastName() const
82 {
83     // Const before return type prevents client from modifying
84     // private data. Client should copy returned string before
85     // destructor deletes storage to prevent undefined pointer.
86     return lastName;
87 }

```



Outline

1.2 Function definitions

```

88 // Fig. 17.9: fig17_09.cpp
89 // Driver to test the employee class
90 #include <iostream>
91
92 using std::cout;
93 using std::endl;
94
95 #include "employ1.h"
96
97 int main()
98 {
99     cout << "Number of employees before instantiation is "
100         << Employee::getCount() << endl;    // use class name
101
102     Employee *e1Ptr = new Employee( "Susan", "Baker" );
103     Employee *e2Ptr = new Employee( "Robert", "Jones" );
104
105     cout << "Number of employees after instantiation is "
106         << e1Ptr->getCount();
107
108     cout << "\n\nEmployee 1: "
109         << e1Ptr->getFirstName()
110         << " " << e1Ptr->getLastName()
111         << "\nEmployee 2: "
112         << e2Ptr->getFirstName()
113         << " " << e2Ptr->getLastName() << "\n\n";
114
115     delete e1Ptr;    // recapture memory
116     e1Ptr = 0;
117     delete e2Ptr;    // recapture memory
118     e2Ptr = 0;

```



Outline



1. Initialize objects
2. Function calls
3. Print data

```
119
120     cout << "Number of employees after deletion is "
121         << Employee::getCount() << endl;
122
123     return 0;
124 }
```



Outline

4. Output

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2
```

```
Employee 1: Susan Baker
Employee 2: Robert Jones
```

```
~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

17.8 Data Abstraction and Information Hiding

- Information hiding
 - Classes hide implementation details from clients
 - Example: stack data structure
 - Data elements like a pile of dishes - added (pushed) and removed (popped) from top
 - Last-in, first-out (LIFO) data structure
 - Client does not care how stack is implemented, only wants LIFO data structure



17.8 Data Abstraction and Information Hiding (II)

- Abstract data types (ADTs)
 - Model real world objects
 - **int**, **float** are models for a numbers
 - Imperfect - finite size, precision, etc.
- C++ an extensible language
 - Base cannot be changed, but new data types can be created



17.8.1 Example: Array Abstract Data Type

- Array
 - Essentially a pointer and memory locations
- Programmer can make an ADT array
 - New capabilities
 - Subscript range checking, array assignment and comparison, dynamic arrays, arrays that know their sizes...
- New classes
 - Proprietary to an individual, to small groups or to companies, or placed in standard class libraries



17.8.2 Example: String Abstract Data Type

- C++ intentionally sparse
 - Reduce performance burdens
 - Use language to create what you need, i.e. a **string** class
- **string** not a built-in data type
 - Instead, C++ enables you to create your own **string** class



17.8.3 Example: Queue Abstract Data Type

- Queue - a waiting line
 - Used by computer systems internally
 - We need programs that simulate queues
- Queue has well-understood behavior
 - Enqueue - put things in a queue one at a time
 - Dequeue - get those things back one at a time on demand
 - Implementation hidden from clients
- Queue ADT - stable internal data structure
 - Clients may not manipulate data structure directly
 - Only queue member functions can access internal data



17.9 Container Classes and Iterators

- Container classes (collection classes)
 - Classes designed to hold collections of objects
 - Services such as insertion, deletion, searching, sorting, or testing an item
 - Examples:
 - Arrays, stacks, queues, trees and linked lists
- Iterator objects (iterators)
 - Object that returns the next item of a collection (or some action)
 - Can have several iterators per container
 - Book with multiple bookmarks
 - Each iterator maintains its own “position” information

