More C++ Classes

C++ Classes

- Preprocessor Wrapper
- Time Class Case Study
- Class Scope and Assessing Class Members
  - Using handles to access class members
- Access and Utility Functions
- Destructors
- Calling Constructors and Destructors

20.2 Preprocesor Wrappers

- Prevent code from being included more than once.
  - #ifndef – "if not defined"
    - Skip this code if it has been included already
  - #define
    - Define a name so this code will not be included again
  - #endif
- If the header has been included previously
  - Name is defined already and the header file is not included again.
- Prevents multiple-definition errors
- Example
  - ```
    #ifndef TIME_H
    #define TIME_H
    … // code
    #endif
    ```
    ```

Time.h with Preprocessor Wrapper

```
1  // Fig. 20.1: Time.h
2  // Declaration of class Time.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    Time(); // constructor
14    void setTime( int, int, int ); // set hour, minute and second
15    void printUniversal(); // print time in universal-time format
16    void printStandard(); // print time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Preprocessor directive **#ifndef** determines whether a name is defined

Preprocessor directive **#define** defines a name (e.g., **TIME_H**)

Preprocessor directive **#endif** marks the end of the code that should not be included multiple times

Time Class Case Study

```
1  // Fig. 20.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero.
13 // Ensures all Time objects start in a consistent state.
14 Time::Time()
15 {
16    hour = minute = second = 0;
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
21 void Time::setTime( int h, int m, int s )
22 {
23    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
24    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
25    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
26 } // end function setTime
```

Ensure that **hour**, **minute** and **second** values remain valid

Time Class Case Study

```
27
28 // print Time in universal-time format (HH:MM:SS)
29 void Time::printUniversal()
30 {
31    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
32       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
33 } // end function printUniversal
34
35 // print Time in standard-time format (HH:MM:SS AM or PM)
36 void Time::printStandard()
37 {
38    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
39       << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
40       << second << ( hour < 12 ? " AM" : " PM" );
41 } // end function printStandard
```

Using `setfill` stream manipulator to specify a fill character

Time Class Case Study

```
1 // Fig. 20.3: fig20_03.cpp
2 // Program to test class Time.
3 // NOTE: This file must be compiled with Time.cpp.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // include definition of class Time from Time.h
9
10 int main()
11 {
12    Time t; // instantiate object t of class Time
13
14    // output Time object t's initial values
15    cout << "The initial universal time is ";
16    t.printUniversal(); // 00:00:00
17    cout << "\nThe initial standard time is ";
18    t.printStandard(); // 12:00:00 AM
19
20    t.setTime( 13, 27, 6 ); // change time
21
22    // output Time object t's new values
23    cout << "\n\nUniversal time after setTime is ";
24    t.printUniversal(); // 13:27:06
25    cout << "\nStandard time after setTime is ";
26    t.printStandard(); // 1:27:06 PM
27
28    t.setTime( 99, 99, 99 ); // attempt invalid settings
```

Time Class Case Study

```
29
30    // output t's values after specifying invalid values
31    cout << "\n\nAfter attempting invalid settings:"
32       << "\nUniversal time: ";
33    t.printUniversal(); // 00:00:00
34    cout << "\nStandard time: ";
35    t.printStandard(); // 12:00:00 AM
36    cout << endl;
37    return 0;
38 } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Common Programming Error 20.1

- Attempting to initialize a **non-static** data member of a class explicitly in the class definition is a syntax error.

20.2 Time Class Case Study

- Parameterized stream manipulator **setfill**
  - Specifies the fill character
    - Which is displayed when an output field wider than the number of digits in the output value.
    - By default, fill characters appear to the left of the digits in the number.
  - **setfill** is a "sticky" setting
    - Applies for all subsequent values that are displayed in fields wider than the value being displayed.

Time Class Case Study

- Member function declared in a class definition but defined outside that class definition
  - Are still within the class's scope.
  - Are known only to other members of the class unless referred to via:
    - Object of the class
    - Reference to an object of the class
    - Pointer to an object of the class
    - Binary scope resolution operator
- Member function defined in the body of a class definition
  - C++ compiler attempts to inline calls to the member function.

Time Class Case Study

- Using class Time
  - Once class Time has been defined, it can be used in declarations:
    - Time sunset;
    - Time arrayOfTimes[5];
    - Time &dinnerTime = sunset;
    - Time *timePtr = &dinnerTime;

Performance Tip 20.2

- Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator **sizeof** to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable (also called **reentrant code** or **pure procedure**) and, hence, can be shared among all objects of one class.

20.3 Class Scope and Accessing Class Members

- Class scope contains
  - Data members
    - Variables declared in the class definition.
  - Member functions
    - Functions declared in the class definition.
- Nonmember functions are defined at file scope.

Class Scope and Accessing Class Members

- Within a class's scope
  - Class members are accessible by all member functions.
- Outside a class's scope
  - public class members are referenced through a handle:
    - An object name
    - A reference to an object
    - A pointer to an object.

Class Scope and Accessing Class Members

- Variables declared in a member function
  - Have block scope.
  - Are known only to that function.
- Hiding a class-scope variable
  - In a member function, define a variable with the same name as a variable with class scope.
  - Such a hidden variable can be accessed by preceding the name with the class name followed by the scope resolution operator (`::`)

Class Scope and Accessing Class Members

- Dot member selection operator (.)
  - Accesses the object's members.
  - Used with an object's name or with a reference to an object.
- Arrow member selection operator (->)
  - Accesses the object's members.
  - Used with a pointer to an object.

Using Handles to Access Class Members

```
1  // Fig. 20.4: fig20_04.cpp
2  // Demonstrating the class member access operators . and ->
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // class Count definition
8  class Count
9  {
10 public: // public data is dangerous
11    // sets the value of private data member x
12    void setX( int value )
13    {
14       x = value;
15    } // end function setX
16
17    // prints the value of private data member x
18    void print()
19    {
20       cout << x << endl;
21    } // end function print
22
23 private:
24    int x;
25 }; // end class Count
```

Using Handles to Access Class Members

```
26
27 int main()
28 {
29    Count counter; // create counter object
30    Count *counterPtr = &counter; // create pointer to counter
31    Count &counterRef = counter; // create reference to counter
32
33    cout << "Set x to 1
34    counter.setX( 1 ); // set data member x to 1
35    counter.print(); // call member function print
36
37    cout << "Set x to 2 and
38    counterRef.setX( 2 ); // set data member x to 2
39    counterRef.print(); // call member function print
40
41    cout << "Set x to 3 and print using a pointer to an object: ";
42    counterPtr->setX( 3 ); // set data member x to 3
43    counterPtr->print(); // call member function print
44    return 0;
45 } // end main
```

Using the dot member selection operator with an object

Using the dot member selection operator with a reference

Using the arrow member selection operator with a pointer

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

20.5 Access and Utility Functions

- Access functions
  - Can read or display data.
  - Can test the truth or falsity of conditions.
    - Such functions are often called **predicate functions.**
    - For example, **isEmpty** function for a class capable of holding many objects.
    - Namely, a **container class** holding many objects such as a linked list, stack or queue.
- Utility functions (also called **helper functions**)
  - **private** member functions that support the operation of the class's **public** member functions.
  - Not part of a class's **public** interface
    - Not intended to be used by clients of a class.

Access and Utility Functions Example

```
1  // Fig. 20.5: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson
8  {
9  public:
10    SalesPerson(); // constructor
11    void getSalesFromUser(); // input sales from keyboard
12    void setSales( int, double ); // set sales for a specific month
13    void printAnnualSales(); // summarize and print sale
14 private:
15    double totalAnnualSales(); // prototype for utility function
16    double sales[ 12 ]; // 12 monthly sales figures
17 }; // end class SalesPerson
18
19 #endif
```

Prototype for a `private` utility function

Access and Utility Functions Example

```
1  // Fig. 20.6: SalesPerson.cpp
2  // Member functions for class SalesPerson.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include "SalesPerson.h" // include SalesPerson class definition
13
14 // initialize elements of array sales to 0.0
15 SalesPerson::SalesPerson()
16 {
17    for ( int i = 0; i < 12; i++ )
18       sales[ i ] = 0.0;
19 } // end SalesPerson constructor
```

Access and Utility Functions Example

```
20
21 // get 12 sales figures from the user at the keyboard
22 void SalesPerson::getSalesFromUser()
23 {
24    double salesFigure;
25
26    for ( int i = 1; i <= 12; i++ )
27    {
28       cout << "Enter sales amount for month " << i << ": ";
29       cin >> salesFigure;
30       setSales( i, salesFigure );
31    } // end for
32 } // end function getSalesFromUser
33
34 // set one of the 12 monthly sales figures; function subtracts
35 // one from month value for proper subscript in sales array
36 void SalesPerson::setSales( int month, double amount )
37 {
38    // test for valid month and amount values
39    if ( month >= 1 && month <= 12 && amount > 0 )
40       sales[ month - 1 ] = amount; // adjust for subscripts 0-11
41    else // invalid month or amount value
42       cout << "Invalid month or sales figure" << endl;
43 } // end function setSales
```

Access and Utility Functions Example

```
44
45 // print total annual sales (with the help of utility function)
46 void SalesPerson::printAnnualSales()
47 {
48    cout << setprecision( 2 ) << fixed
49       << "\nThe total annual sales are: $"
50       << totalAnnualSales() << endl; // call utility function
51 } // end function printAnnualSales
52
53 // private utility function to total annual sales
54 double SalesPerson::totalAnnualSales()
55 {
56    double total = 0.0; // initialize total
57
58    for ( int i = 0; i < 12; i++ ) // summarize sales results
59       total += sales[ i ]; // add month i sales to total
60
61    return total;
62 } // end function totalAnnualSales
```

Calling a `private` utility function

Definition of a `private` utility function

Access and Utility Functions Example

```
1  // Fig. 20.7: fig20_07.cpp
2  // Demonstrating a utility function.
3  // Compile this program with SalesPerson.cpp
4
5  // include SalesPerson class definition from SalesPerson.h
6  #include "SalesPerson.h"
7
8  int main()
9  {
10    SalesPerson s; // create SalesPerson object s
11
12    s.getSalesFromUser(); // note simple sequential code;
13    s.printAnnualSales(); // no control statements in main
14    return 0;
15 } // end main
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

20.7 Destructors

- A special member function
- The destructor name is the tilde character (~) followed by the class name, e.g., `~Time`.
- Is called implicitly when an object is destroyed.
  - For example, this occurs as an automatic object is destroyed when program execution leaves the scope in which that object was instantiated.
- A destructor does not release the object's memory.
  - It performs termination housekeeping.
  - The system reclaims the object's memory.
    - So the memory may be reused to hold new objects.

Destructors

- Destructor
  - Receives no parameters and returns no value.
    - Cannot specify a return type—not even **void**!
  - A class may have only one destructor.
    - Destructor overloading is not allowed.
  - If the programmer does not explicitly provide a destructor, the compiler creates an "empty" destructor.

20.8 Calling Constructors and Destructors

- Constructors and destructors
  - Are called implicitly by the compiler.
    - The order of these function calls depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
  - Generally,
    - Destructor calls are made in the reverse order of the corresponding constructor calls.
  - However,
    - Storage classes of objects can alter the order in which destructors are called.

Calling Constructors and Destructors

- For objects defined in global scope, constructors are called before any other function (including **main**) in that file begins execution.
- The corresponding destructors are called when **main** terminates.
- The function **exit**
  - Forces a program to terminate immediately.
  - Does not execute the destructors of automatic objects.
  - Is often used to terminate a program when an error is detected.
- The function **abort**
  - Performs similarly to function **exit**.
  - But it forces the program to terminate immediately without allowing the destructors of any objects to be called.
  - Usually used to indicate an abnormal termination of the program.

Calling Constructors and Destructors

- For an automatic local object
  - A constructor is called when that object is defined.
  - The corresponding destructor is called when execution leaves the object's scope.
- For automatic objects
  - Constructors and destructors are called each time execution enters and leaves the scope of the object.
  - Automatic object destructors are not called if the program terminates with an **exit** or **abort** function.

Calling Constructors and Destructors

- For a **static** local object
  - The constructor is called only once –
    - When execution first reaches where the object is defined.
  - The destructor is called when **main** terminates or the program calls function **exit**.
  - Destructor is not called if the program terminates with a call to function **abort**.
- **Global** and **static** objects are destroyed in the reverse order of their creation.

Destructor Example

```
1  // Fig. 20.11: CreateAndDestroy.h
2  // Definition of class CreateAndDestroy.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using std::string;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

Destructor Example

```
1  // Fig. 20.12: CreateAndDestroy.cpp
2  // Member-function definitions for class CreateAndDestroy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
8
9  // constructor
10 CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
11 {
12    objectID = ID; // set object's ID number
13    message = messageString; // set object's descriptive message
14
15    cout << "Object " << objectID << "   constructor runs   "
16       << message << endl;
17 } // end CreateAndDestroy constructor
18
19 // destructor
20 CreateAndDestroy::~CreateAndDestroy()
21 {
22    // output newline for certain objects; helps readability
23    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
24
25    cout << "Object " << objectID << "   destructor runs    "
26       << message << endl;
27 } // end ~CreateAndDestroy destructor
```

Defining the class's destructor

Destructor Example

```
1  // Fig. 20.13: fig20_13.cpp
2  // Demonstrating the order in which constructors and
3  // destructors are called.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
9
10 void create( void ); // prototype
11
12 CreateAndDestroy first( 1, "(global before main)" ); // global object
13
14 int main()
15 {
16    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
17    CreateAndDestroy second( 2, "(local automatic in main)" );
18    static CreateAndDestroy third( 3, "(local static in main)" );
19
20    create(); // call function to create objects
21
22    cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
23    CreateAndDestroy fourth( 4, "(local automatic in main)" );
24    cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
25    return 0;
26 } // end main
```

Object created outside of `main`

Local automatic object created in `main`

Local `static` object created in `main`

Local automatic object created in `main`

Destructor Example

```
27
28 // function to create objects
29 void create( void )
30 {
31    cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
32    CreateAndDestroy fifth( 5, "(local automatic in create)" );
33    static CreateAndDestroy sixth( 6, "(local st
34    CreateAndDestroy seventh( 7, "(local automatic in create)" );
35    cout << "\nCREATE FUNCTION: EXECUTION ENDS" <<
36 } // end function create
```

Local automatic object created in `create`

Local `static` object created in `create`

Local automatic object created in `create`

Destructor Example

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

20.10 Default Memberwise Assignment

- Default memberwise assignment
  - Assignment operator (=)
    - Can be used to assign an object to another object of the same type.
      - Each data member of the right object is assigned to the same data member in the left object.
    - Can cause serious problems when **data members contain pointers to dynamically allocated memory !!**

20.10 Default Memberwise Assignment

```
1  // Fig. 20.17: Date.h
2  // Declaration of class Date.
3  // Member functions are defined in Date.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef DATE_H
7  #define DATE_H
8
9  // class Date definition
10 class Date
11 {
12 public:
13    Date( int = 1, int = 1, int = 2000 ); // default constructor
14    void print();
15 private:
16    int month;
17    int day;
18    int year;
19 }; // end class Date
20
21 #endif
```

Default initialization of data members

20.10 Default Memberwise Assignment

```
1  // Fig. 20.18: Date.cpp
2  // Member-function definitions for class Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // include definition of class Date from Date.h
8
9  // Date constructor (should do range checking)
10 Date::Date( int m, int d, int y )
11 {
12    month = m;
13    day = d;
14    year = y;
15 } // end constructor Date
16
17 // print Date in the format mm/dd/yyyy
18 void Date::print()
19 {
20    cout << month << '/' << day << '/' << year;
21 } // end function print
```

20.10 Default Memberwise Assignment

```
1  // Fig. 20.19: fig20_19.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using default memberwise assignment.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Date.h" // include definition of class Date from Date.h
9
10 int main()
11 {
12    Date date1( 7, 4, 2004 );
13    Date date2; // date2 defaults to 1/1/2000
14
15    cout << "date1 = ";
16    date1.print();
17    cout << "\ndate2 = ";
18    date2.print();
19
20    date2 = date1; // default memberwise assignment
21
22    cout << "\n\nAfter default memberwise assignment, date2 = ";
23    date2.print();
24    cout << endl;
25    return 0;
26 } // end main
```

memberwise assignment assigns data members of date1 to date2

date2 now stores the same date as date1

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

20.10 Default Memberwise Assignment

- **Copy constructor**
  - Enables pass-by-value for objects
    - Used to copy original object's values into new object to be passed to a function or returned from a function
  - Compiler provides a **default copy constructor**
    - Copies each member of the original object into the corresponding member of the new object (i.e., memberwise assignment).
  - Can cause serious problems when **data members contain pointers to dynamically allocated memory!!**

Summary

- Preprocessor Wrapper Example
- Time Class Case Study
  - Setfill, setw, conditional
- Class Scope and Assessing Class
  - Handles
  - Members
  - Using handles to access class members
  - Dot and arrow member selection operators.
- Access and Utility Functions
  - Predicates, container classes and helpers
- Destructors
  - Restrictions, default, order of execution
  - Functions exit and abort
- Calling Constructors and Destructors
  - Differences between static and automatic objects
  - Destructor Example
- Default Memberwise Assignment
  - Default Copy Constructor