Object-Oriented Programming

Lecture 7: Classes - Additional

Agenda

- Class interface separation
- Compilation and link process
- Class scope and accessing class member
- Access functions and utility functions
- Constructors with default arguments
- Destructors

Separating Interface from Implementation

- The use an object of a class, the client code (e.g., main) needs to know only
    - what member functions to call
    - what arguments to provide to each member function, and
    - what return type to expect from each member function.
    - The client code does not need to know how those functions are implemented.
    - If client code does know how a class is implemented, the programmer might write client code based on the class's implementation details.

“time.h”

```
#include <string>

#ifndef TIME_H  // Prevent multiple inclusions
#define TIME_H

class Time {
public:
   void setTime(int hour, int minute, int second);  // Set hour, minute, and second
   std::string toUniversalString() const;  // Return 24-hour time format string
   std::string toStandardString() const;   // Return 12-hour time format string

private:
   unsigned int hour{0};   // 0 - 23 (24-hour clock format)
   unsigned int minute{0}; // 0 - 59
   unsigned int second{0}; // 0 - 59
};

#endif
```

- Use #ifndef, #define and #endif preprocessing directives to form an include guard that prevents headers from being included more than once in a source-code file.
- By convention, use the name of the header in uppercase with the period replaced by an underscore in the #ifndef and #define preprocessing directives of a header.

“time.cpp”

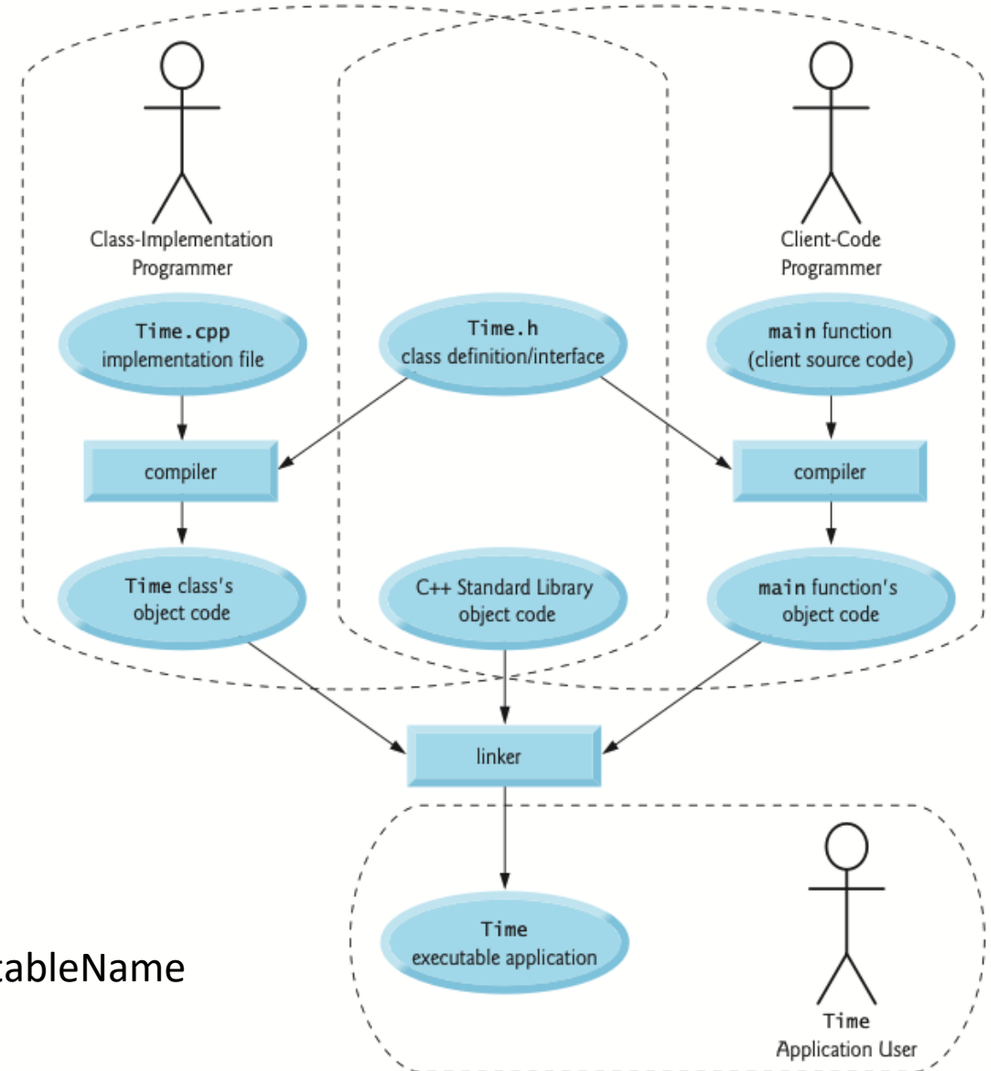
```
#include <iomanip> // For setw and setfill stream manipulators
#include <stdexcept> // For invalid_argument exception class
#include <sstream> // For ostringstream class
#include <string>
#include "time.h" // Include definition of class Time from Time.h

using namespace std;

// Set new Time value using universal time
void Time::setTime(int h, int m, int s) {
   // Validate hour, minute, and second
   if (h >= 0 && h < 24 && m >= 0 && m < 60 && s >= 0 && s < 60) {
      hour = h;
      minute = m;
      second = s;
   } else {
      throw invalid_argument("hour, minute and/or second was out of range");
   }
}

// Return Time as a string in universal-time format (HH:MM:SS)
string Time::toUniversalString() const {
   ostringstream output;
   output << setfill('0') << setw(2) << hour << ":"
      << setw(2) << minute << ":" << setw(2) << second;
   return output.str(); // Returns the formatted string
}
```

```
// Return Time as a string in standard-time format (HH:MM:SS AM or PM)
string Time::toStandardString() const {
   ostringstream output;
   output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
      << setfill('0') << setw(2) << minute << ":"
      << setw(2) << second << (hour < 12 ? " AM" : " PM");
   return output.str(); // Returns the formatted string
}
```

“main.c”

```
#include <iostream>
#include <stdexcept> // For invalid_argument exception class
#include "time.h" // Definition of class Time from Time.h

using namespace std;

// Displays a Time in 24-hour and 12-hour formats
void displayTime(const string& message, const Time& time) {
   cout << message
        << "\nUniversal time: " << time.toUniversalString()  // Corrected
        << "\nStandard time: " << time.toStandardString()  // Corrected
        << "\n\n";
}

int main() {
   Time t; // Instantiate object t of class Time
   displayTime("Initial time:", t); // Display t's initial value

   t.setTime(13, 27, 6); // Change time
   displayTime("After setTime:", t); // Display t's new value

   // Attempt to set the time with invalid values
   try {
      t.setTime(99, 99, 99); // All values out of range
   } catch (invalid_argument& e) {
      cout << "Exception: " << e.what() << "\n\n";
   }

   // Display t's value after attempting to set an invalid time
   displayTime("After attempting to set an invalid time:", t);

   return 0; // Added return statement
}
```

Compilation and Linking Process



g++ -std=c++14 *.cpp -o ExecutableName

Class Scope

- **Class Members:** Within a class, both data members (variables) and member functions (methods) are defined. These are scoped within the class, meaning they are part of the class's namespace and not the global namespace.
- **Access Within Class**: Inside the class, all members are immediately accessible to other members of the class. They can be referenced directly by their names without any special syntax.
- **Non-Member Functions:** Functions not defined inside the class are considered non-member functions and reside in the global namespace, unless specifically placed in another namespace.

Accessing Class Members

- Outside the class scope, accessing class members depends on their access specifiers (public, private, protected):
- To access public class members from outside the class, you can use:
  - Object Name: Using the dot operator (.) with the object name.
  - Reference to Object: Using the dot operator with a reference to the object.
  - Pointer to Object: Using the arrow operator (->) with a pointer to the object.
- These methods (object, reference, pointer) are often referred to as "handles" to the object. The type of handle determines what members are accessible.

Dot (.) and Arrow (->) Operators

- Dot Operator (.): Used with an object name or a reference to an object to access its members. Syntax: objectName.memberName or referenceName.memberName.
- Arrow Operator (->): Used with a pointer to an object to access its members. Syntax: pointerName->memberName.

```cpp
#include <iostream>

// Definition of the Account class
class Account {
public:
    // Data member to store the account balance
    double balance;

    // Constructor to initialize the account balance
    Account(double initialBalance) : balance(initialBalance) {}

    // Member function to set the account balance
    void setBalance(double newBalance) {
        if (newBalance >= 0.0) {
            balance = newBalance;
        } else {
            std::cout << "Balance cannot be negative." << std::endl;
        }
    }

    // Member function to display the account balance
    void displayBalance() const {
        std::cout << "Account balance: $" << balance << std::endl;
    }
};

int main() {
    // Create an Account object with an initial balance
    Account account(100.0); // An Account object with initial balance 100.0

    // Reference to the Account object
    Account& accountRef = account;

    // Pointer to the Account object
    Account* accountPtr = &account;

    // Call setBalance via the Account object
    account.setBalance(123.45);
    account.displayBalance(); // Display the balance to verify the change

    // Call setBalance via a reference to the Account object
    accountRef.setBalance(200.00);
    accountRef.displayBalance(); // Display the balance to verify the change

    // Call setBalance via a pointer to the Account object
    accountPtr->setBalance(300.00);
    accountPtr->displayBalance(); // Display the balance to verify the change

    return 0;
}
```

Access Functions

- Used within a class to read or display the class's data members without modifying them.
- These functions provide a controlled way to access the internal state of an object.
- Predicate functions are a specific type of access functions that return a boolean value, often used to check conditions or states of an object, like whether a container is empty.

```
class Stack {
private:
    static const int SIZE = 10; // Size of the stack
    int data[SIZE]; // Array to store stack elements
    int top; // Index of the top element

public:
    Stack() : top(-1) {} // Constructor initializes the stack to be empty

    // Access function to check if the stack is empty
    bool isEmpty() const {
        return top == -1;
    }

    // Other member functions...
};
```

Utilitiy (Helper) Functions

- Private member functions designed to support the operations of other member functions within a class.
- They are not intended to be accessed directly by the class's clients (users of the class).
- Utility functions help avoid code duplication by encapsulating common operations used by several member functions.

```
class Stack {
private:
   static const int SIZE = 10;
   int data[SIZE];
   int top;

   // Utility function to check if the stack is full
   bool isFull() const {
      return top == SIZE - 1;
   }

public:
   Stack() : top(-1) {}

   bool isEmpty() const {
      return top == -1;
   }

   // Member function to add an element to the stack
   void push(int value) {
      if (!isFull()) { // Use the utility function to check if the stack is full
         data[++top] = value;
      } else {
         std::cout << "Stack is full, cannot push " << value << std::endl;
      }
   }

   // Other member functions...
};
```

Constructors with Default Arguments

- Constructors can specify default arguments
- **Implicit Conversion:** This occurs when the compiler automatically converts one data type to another. In the context of class objects, this can happen when a constructor is callable with a single argument, allowing an object of one type to be initialized with a value of another type without the programmer explicitly requesting a conversion.
- **Explicit Conversion**: This requires the programmer to clearly specify that a conversion should occur. For class types, this means that the constructor must be called directly or with a cast.

Without explicit Keyword

```
class Time {
public:
    // Constructor without 'explicit' allows implicit conversions
    Time(int hours) { /* implementation */ }
    // Other members...
};

void scheduleMeeting(Time t) { /* implementation */ }

int main() {
    scheduleMeeting(10); // Implicitly converts 10 to a Time object
}
```

Destructors

- Special member function that is automatically called when an object of a class is destroyed.
- This can happen when an object goes out of scope, or when it is explicitly deleted if it was created dynamically using ‘new’.
- The syntax for a destructor is a tilde character (~) followed by the class name, with no parameters and no return type. For example, for a class named Example, the destructor would be declared as ~Example().
- The primary purpose of a destructor is to perform "cleanup" operations required before an object is destroyed.

```
#include <iostream>

class Example {
public:
    Example() { /* Constructor code */ }
    ~Example() {
        // Destructor code: clean up resources, if any
        std::cout << "Destructor called, cleaning up..." << std::endl;
    }
};

int main() {
    Example obj; // Constructor called here
    // obj's scope ends with the function, so destructor called here
}
```

Returning a Reference or a Pointer to a private Data Member

- Returning a reference or a pointer to a private data member from a member function can be a powerful feature.
- But it should be used with caution due to the potential risks it introduces in terms of encapsulation and data integrity.
- Implications and Considerations:
    - Encapsulation Breach
    - Life-time Management
    - Const Correctness

```cpp
#include <iostream>

class Vector {
private:
    double x, y, z; // Private data members

public:
    // Constructor to initialize the vector
    Vector(double xVal, double yVal, double zVal) : x(xVal), y(yVal), z(zVal) {}

    // Member function returning a reference to a private member
    double& getX() { return x; }

    // Member function returning a pointer to a private member
    double* getYPtr() { return &y; }

    // Const member function for read-only access to a private member
    const double& getZ() const { return z; }
};

int main() {
    Vector vec(1.0, 2.0, 3.0);

    // Modify x directly through its reference
    double& xRef = vec.getX();
    xRef = 4.0;

    // Access y using a pointer
    double* yPtr = vec.getYPtr();
    *yPtr = 5.0;

    // Read z through a const reference (cannot modify z)
    const double& zRef = vec.getZ();
    std::cout << "Vector: (" << vec.getX() << ", " << *vec.getYPtr() << ", " << zRef
<< ")" << std::endl;

    return 0;
}
```

- The Vector class has three private double variables: x, y, and z.
- getX() returns a non-const reference to x, allowing for modification of x.
- getYPtr() returns a pointer to y, allowing both read and write access to y.
- getZ() returns a const reference to z, allowing read-only access to z and ensuring z cannot be modified through this reference.
- In the main function, we demonstrate modifying x and y through the reference and pointer obtained from getX() and getYPtr(), respectively. z is accessed for reading through a const reference.

lec09_04.cpp

Default Memberwise Assignment

- Refers to the behavior of the assignment operator (=) when used with objects of user-defined types (classes).
- When one object is assigned to another object of the same class using the assignment operator and no custom assignment operator is defined for that class, C++ performs what is known as a memberwise assignment or copy assignment.
- This means that each data member of the source object (the object on the right-hand side of the assignment operator) is copied to the corresponding data member of the destination object (the object on the left-hand side of the assignment operator).

lec09_05.cpp

```cpp
#include <iostream>

class Date {
public:
    int day;
    int month;
    int year;

    // Constructor to initialize the Date object
    Date(int d, int m, int y) : day(d), month(m), year(y) {}

    // Function to display the date
    void display() const {
        std::cout << day << "/" << month << "/" << year << std::endl;
    }
};

int main() {
    // Create and initialize Date object date1
    Date date1(15, 10, 2021);
    std::cout << "date1: ";
    date1.display();

    // Create another Date object date2 with a different date
    Date date2(1, 1, 2020);
    std::cout << "date2 before assignment: ";
    date2.display();

    // Use default memberwise assignment to assign date1 to date2
    date2 = date1;

    // Display date2 after assignment
    std::cout << "date2 after assignment: ";
    date2.display();

    return 0;
}
```

‘const’ Objects

- When an object is declared as const, it means the object is not modifiable after initialization. Any attempt to change the state of a const object will result in a compilation error.
- For example, const Time noon{12, 0, 0}; declares a const object noon of a class Time and initializes it to represent 12:00 noon. This object cannot be modified after its declaration.

‘const’ Member Functions

- C++ requires that any member function called on a const object must also be declared as const. This ensures that the member function does not modify the object.
- Declaring a member function as const is a promise that the function does not alter the state of the object, making it safe to call on const objects.
- Even accessor functions (getters), which do not modify the object, must be declared const to be callable on const objects.

Constructors, Destructors, and ‘const’ Objects

- Constructors and destructors are special member functions that are allowed to modify objects. Constructors initialize objects, and destructors perform cleanup before the object's memory is reclaimed.
- It's a compilation error to declare constructors and destructors as const because their primary purpose is to modify the object's state (initialization and cleanup).
- The "constness" of a const object is enforced from the end of the constructor's execution until the destructor is called.

Composition: Objects as Members of Classes

- **Composition** is a fundamental design principle where one class contains objects of another class as its members, establishing a "has-a" relationship.
- This section outlines how this principle enables software reuse, enhances design clarity, and facilitates the construction of complex objects from simpler ones.
  - Composition involves including instances of one or more classes as member variables within another class.
  - This models a "has-a" relationship, where the containing class (composite) has one or more instances of other classes (components) as part of its state.
  - Composition allows for software reuse by enabling classes to be built using existing classes.

lec09_07.cpp

Static Class Members

- Shared Among Instances: Unlike regular data members of a class where each object has its own copy, a static data member is shared among all instances of the class. There's only one copy of a static member that exists, regardless of how many objects of the class are created.
- Classwide Information: Static members represent information that is not tied to a specific instance but rather belongs to the class as a whole.
- This is useful for constants, counters, or other shared data that should remain consistent across all instances of the class.

Accessing Static Members

- Through Public Member Functions: Private or protected static members are typically accessed via public member functions or friend functions/classes. This encapsulates the static member, adhering to the principles of encapsulation and data hiding.

- Without Objects: Since static members are not tied to specific instances, they can be accessed without any objects of the class existing. For public static members, you can use the class name with the scope resolution operator (::) to access them (e.g., Martian::martianCount).

- Static Member Functions: Similar to static data members, static member functions are not associated with any particular object. They can be called using the class name and the scope resolution operator, and they can only access static data members or other static member functions.

lec09_08.cpp

Matrix operation

Transpose
https://www.youtube.com/watch?v=kLbieVyn41o

Multiplication
https://www.youtube.com/watch?v=P5GJJ02OG08

Git Tutorial
https://www.youtube.com/watch?v=tRZGeaHPoaw

SFML (Simple and Fast Multimedia Library)
https://www.youtube.com/playlist?list=PL6xSOsbVA1eb_QqMTTcql_3PdOiE928up

QT C++ GUI
https://www.youtube.com/watch?v=70zn9jA9tZ0&list=PLQMs5svASiXMUlkVnxPGWz9qoC2YOO9OV

Q & A