C++ Basics
Classes

Separating Interface from Implementation

Conventional software engineering wisdom says that to 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.

Separating Interface from Implementation

- The client code <u>does not need to know</u> how those functions are **implemented**.
- If client code *does* know how a class is implemented, the programmer <u>might write client code based on the class's implementation details</u>.
- Ideally, if that implementation <u>changes</u>, the class's <u>**clients** should not have to change</u>.

➢ *Hiding the class's implementation details makes it easier to change the class's implementation* while minimizing, and hopefully eliminating, changes to client code.

Interface of a Class

Interfaces define and standardize the ways in which things such as people and systems **interact** with one another.

The interface of a class describes **what services a class's clients can use** and **how to request those services**, but **not** how the class carries out the services.

A class's public interface consists of the class's public member functions (also known as the class's public services).

Separating Interface from Implementation

To **separate** the class's interface from its implementation, we break up class Time into two files - the header **Time.h** in which **class Time is defined**, and the source-code file **Time.cpp** in which Time's **member functions are defined** - so that

1. the class is reusable,

2. the clients of the class know what member functions the class provides, how to call them and what return types to expect, and

3. the clients **do not know** how the class's member functions are implemented.

By convention, member-function definitions are placed in a source-code file of the same base name (e.g., Time) as the class's header but with a **.cpp** filename extension (some compilers support other filename extensions as well)

Time.h - **header file**

Time.cpp (.C, .cxx) – **source-code file**

Time class

```
4   #include <string>
5
6   // prevent multiple inclusions of header
7   #ifndef TIME_H
8   #define TIME_H
9
10  // Time class definition
11  class Time {
12  public:
13     void setTime(int, int, int); // set hour, minute and second
14     std::string toUniversalString() const; // 24-hour time format string
15     std::string toStandardString() const; // 12-hour time format string
16  private:
17     unsigned int hour{0}; // 0 - 23 (24-hour clock format)
18     unsigned int minute{0}; // 0 - 59
19     unsigned int second{0}; // 0 - 59
20  };
21
22  #endif
```

Lines 13–15 *describe the class's public interface <u>without revealing the member-function implementations</u>.*

Time class

The class definition is enclosed in the following **include guard:**

#ifndef TIME_H
#define TIME_H
...
#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:
  FILENAME.h - > FILENAME_H

- The compiler must know the data members of the class **to determine how much memory to reserve for each object of the class**. Including the header Time.h in the client code provides the compiler with the information it needs to ensure that the client code calls the member functions of class Time correctly.

Time class

```
#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;
```

Time class

```
   output << setfill('0') << setw(2) << hour << ":"
      << setw(2) << minute << ":" << setw(2) << second;
   return output.str(); // returns the formatted string
}

// return Time as 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
}
```

- Objects of class **ostringstream** (from the header <sstream>) provide the same functionality as **cout** <u>but write their output to string objects in memory</u>.

- You use class ostringstream’s **str** member function to get the formatted string.

- Once the fill character is specified with **setfill**, it applies for <u>all subsequent values</u> that are displayed in fields wider than the value being displayed— setfill is a **sticky** setting.

Time class

```
#include <iostream>
#include <stdexcept> // 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()
      << "\nStandard time: " << time.toStandardString() << "\n\n";
}

int main() {
   Time t; // instantiate object t of class Time

   displayTime("Initial time:", t); // display t's initial value
```

Time class

```
    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);
}
```

```
Initial time:
Universal time: 00:00:00
Standard time: 12:00:00 AM

After setTime:
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting to set an invalid time:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Time class

Each member function's name is preceded by the class name and the **scope resolution operator (::).**

The Time:: tells the compiler that each member function is within that class's scope and its name is known to other class members.

- When defining a class's member functions outside that class, omitting the class name and scope resolution operator (::) that should precede the function names causes compilation errors.

Time class

To indicate that the member functions in Time.cpp are part of class Time, we must first include the Time.h header.

When compiling Time.cpp, the compiler uses the information in Time.h to ensure that

- the first line of each member function matches its prototype in the Time.h file.
- each member function knows about the class's data members and other member functions.

Implicitly Inlining Member Functions

- Defining a member function inside the class definition **implicitly inlines** the member function (if the compiler chooses to do so). This can improve *performance*.

- Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header, because *every change to the header requires you to recompile every source-code file that's dependent on that header* (a time-consuming task in large systems).

Time class

```
// 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");
   }
}
```

If any of the values is outside its range, setTime **<u>throws</u> an <u>exception</u>** of type **invalid_argument**, which <u>notifies the client code</u> that an invalid argument was received.

The **throw** statement creates a <u>new object of type **invalid_argument**</u>**.**

After the exception object is created, the throw statement <u>immediately terminates function setTime</u> and the <u>exception is returned to the code that attempted to set the time</u>.

Member Functions vs. Global Functions

- Using an object-oriented programming approach often requires <u>fewer arguments when calling functions</u>. This benefit derives from the fact that encapsulating data members and member functions within a class gives the member functions the right to access the data members.

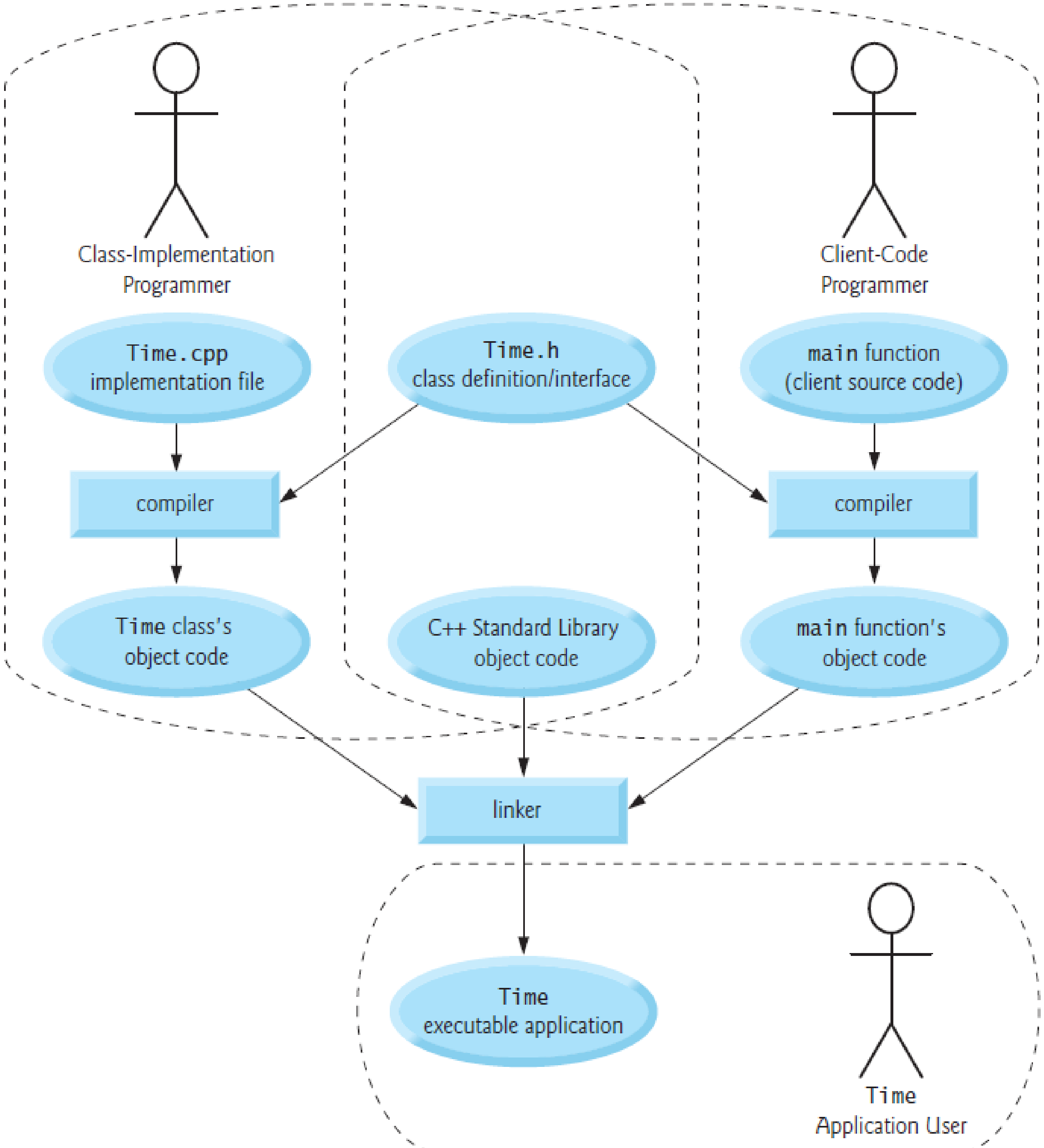
- Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a **constructor** or by **member functions that store new data**.

Object size

- Objects contain **only data**, so objects are much smaller than if they also contained member functions.

- 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 the same for all objects of the class and, hence, can be shared among them.


Class-Implementation Programmer
Client-Code Programmer
Time.cpp implementation file
Time.h class definition/interface
main function (client source code)
compiler
compiler
Time class's object code
C++ Standard Library object code
main function's object code
linker
Time executable application
Time Application User

Scope and Accessing Class Members

Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name.

Outside a class's scope, public class members are referenced through the following **handles** of the object:

- an object name,
- a reference to an object, or
- a pointer to an object

Dot (.) and Arrow (->) Member-Selection Operators are used to access the class members.

Scope and Accessing Class Members

```
Account account; // an Account object
// accountRef refers to an Account object
Account& accountRef{account};
// accountPtr points to an Account object
Account* accountPtr{&account};


// call setBalance via the Account object
account.setBalance(123.45);

// call setBalance via a reference to the Account object
accountRef.setBalance(123.45);

// call setBalance via a pointer to the Account object
accountPtr->setBalance(123.45);
```

Default Constructor

```
Account myAccount;
```

- Here C++ *implicitly* calls the class's **default constructor.**
- In any class that does not explicitly define a constructor, the compiler provides a default constructor with no parameters.
- The default constructor does not initialize the class's fundamental-type data members *but does call the default constructor for each data member that's an object of another class*.
- An uninitialized fundamental-type variable contains an undefined ("garbage") value.
- There's *no default constructor* in a class that defines a constructor.
- *Unless default initialization of your class's data members is acceptable*, you should generally provide a custom constructor to ensure that your data members are properly initialized with meaningful values when each new object of your class is created.

Constructors with Default Arguments

- Like other functions, constructors can specify **default arguments**.

- A constructor **that defaults all its arguments is also a default constructor**— that is, a constructor that can be invoked with no arguments. **There can be at most one default constructor per class**.

- *Any change to the default argument values of a function requires the client code to be* ***recompiled*** *(to ensure that the program still functions correctly).*

Constructors with Default Arguments

```cpp
// prevent multiple inclusions of header
#ifndef TIME_H
#define TIME_H

// Time class definition
class Time {
public:
   explicit Time(int = 0, int = 0, int = 0); // default constructor

   // set functions
   void setTime(int, int, int); // set hour, minute, second
   void setHour(int); // set hour (after validation)
   void setMinute(int); // set minute (after validation)
   void setSecond(int); // set second (after validation)

   // get functions
   unsigned int getHour() const; // return hour
   unsigned int getMinute() const; // return minute
   unsigned int getSecond() const; // return second

   std::string toUniversalString() const; // 24-hour time format string
   std::string toStandardString() const; // 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
```

```cpp
#include <string>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

// Time constructor initializes each data member
Time::Time(int hour, int minute, int second) {
   setTime(hour, minute, second); // validate and set time
}

// set new Time value using universal time
void Time::setTime(int h, int m, int s) {
   setHour(h); // set private field hour
   setMinute(m); // set private field minute
   setSecond(s); // set private field second
}

// set hour value
void Time::setHour(int h) {
   if (h >= 0 && h < 24) {
      hour = h;
   }
   else {
      throw invalid_argument("hour must be 0-23");
   }
}

// set minute value
void Time::setMinute(int m) {
   if (m >= 0 && m < 60) {
      minute = m;
   }
   else {
      throw invalid_argument("minute must be 0-59");
   }
}
```

```cpp
// set second value
void Time::setSecond(int s) {
   if (s >= 0 && s < 60) {
      second = s;
   }
   else {
      throw invalid_argument("second must be 0-59");
   }
}

// return hour value
unsigned int Time::getHour() const {return hour;}

// return minute value
unsigned Time::getMinute() const {return minute;}

// return second value
unsigned Time::getSecond() const {return second;}

// return Time as a string in universal-time format (HH:MM:SS)
string Time::toUniversalString() const {
   ostringstream output;
   output << setfill('0') << setw(2) << getHour() << ":"
      << setw(2) << getMinute() << ":" << setw(2) << getSecond();
   return output.str();
}

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

```cpp
#include "Time.h" // include 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()
      << "\nStandard time: " << time.toStandardString() << "\n\n";
}

```

```cpp
int main() {
   Time t1; // all arguments defaulted
   Time t2{2}; // hour specified; minute and second defaulted
   Time t3{21, 34}; // hour and minute specified; second defaulted
   Time t4{12, 25, 42}; // hour, minute and second specified

   cout << "Constructed with:\n\n";
   displayTime("t1: all arguments defaulted", t1);
   displayTime("t2: hour specified; minute and second defaulted", t2);
   displayTime("t3: hour and minute specified; second defaulted", t3);
   displayTime("t4: hour, minute and second specified", t4);

   // attempt to initialize t5 with invalid values
   try {
      Time t5{27, 74, 99}; // all bad values specified
   }
   catch (invalid_argument& e) {
      cerr << "Exception while initializing t5: " << e.what() << endl;
   }
}
```

```
Constructed with:

t1: all arguments defaulted
Universal time: 00:00:00
Standard time: 12:00:00 AM

t2: hour specified; minute and second defaulted
Universal time: 02:00:00
Standard time: 2:00:00 AM

t3: hour and minute specified; second defaulted
Universal time: 21:34:00
Standard time: 9:34:00 PM

t4: hour, minute and second specified
Universal time: 12:25:42
Standard time: 12:25:42 PM

Exception while initializing t5: hour must be 0-23
```

Notes on calling set/get methods

Time's set and get functions are called throughout the class's body.

In particular, function setTime calls functions setHour, setMinute and setSecond, and functions toUniversalString and toStandardString call functions getHour, getMinute and getSecond.

These functions could have accessed the class's private data directly.

However, consider changing the representation of the time from *three int values* (requiring 12 bytes of memory on systems with four-byte ints) to a **single int value.**

If we made such a change, **only the bodies of the functions that access the private data directly would need to change**.

There would be no need to modify the bodies of functions setTime, toUniversalString or toStandardString.

Notes on calling set/get methods

- If a member function of a class already provides all or part of the functionality required by a constructor or other member functions of the class, call that member function from the constructor or other member functions. ***Avoid repeating code****.*

- A constructor can call other member functions of the class, such as set or get functions, but because the constructor is initializing the object, the data members may not yet be initialized. Using data members before they have been properly initialized can cause logic errors.

- Making data <u>members private and controlling access, especially write access</u>, to those data members through public member functions helps ensure data **integrity**.

- The benefits of data integrity are not automatic simply because data members are <u>made private</u>—you must provide appropriate **validity checking**.

In-class member initializers

In C++98, only **static const members of integral types** could be initialized in-class:

```
int var = 7;
  class X {
     static const int m1 = 7;        // ok
     const int m2 = 7;                    // error: not static
     static int m3 = 7;              // error: not const
     static const int m4 = var;          // error: initializer not constant expression
     static const string m5 = "odd"; // error: not integral type
     // ...
  };
```

In-class member initializers

The basic idea for **C++11** was to allow a non-static data member to be initialized where it is declared (in its class).

```
class A {
  public:
     int a = 7;
};
```

This is equivalent to:

```
class A {
  public:
     int a;
     A() : a(7) {}
};
```

This *saves a bit of typing*, but the real benefits come in classes with multiple constructors.

In-class member initializers

```
class A {
public:
    A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor run") {}
    A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor run") {}
    A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor run") {}
    int a, b;
private:
    HashingFunction hash_algorithm;  // Cryptographic hash to be applied to all A instances
    std::string s;                   // String indicating state in object lifecycle
};
```

In C++11, this will become

```
class A {
public:
    A(): a(7), b(5) {}
    A(int a_val) : a(a_val), b(5) {}
    A(D d) : a(7), b(g(d)) {}
    int a, b;
private:
    HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be applied to all A instances
    std::string s{"Constructor run"};       // String indicating state in object lifecycle
};
```

In-class member initializers

If a member is initialized by both an in-class initializer and a constructor, **only the constructor's initialization is done** (it "overrides" the default). So we can simplify further:

```
class A {
public:
    A() {}
    A(int a_val) : a(a_val) {}
    A(D d) : b(g(d)) {}
    int a = 7;
    int b = 5;
private:
    HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be applied to all A instances
    std::string s{"Constructor run"};       // String indicating state in object lifecycle
};
```

Overloaded Constructors

- A **class's constructors and member functions can also be overloaded**.
- Overloaded constructors typically allow objects to be initialized with <u>different types and/or numbers of arguments</u>.

```
explicit Time(int = 0, int = 0, int = 0);
```

Instead, we could have defined 4 constructors:

```
Time(); // default hour, minute and second to 0
explicit Time(int); // init hour; default minute and second to 0
Time(int, int); // initialize hour and minute; default second to 0
Time(int, int, int); // initialize hour, minute and second
```

C++11 Delegating Constructors

C++11 allows constructors to call other constructors in the same class.

The calling constructor is known as a **delegating constructor**—it *delegates its work to another constructor.*

This is useful when overloaded constructors have common code that previously would have been defined in a private utility function and called by all the constructors.

```
Time::Time() : Time{0, 0, 0} {} // delegate to Time(int, int, int)

// delegate to Time(int, int, int)
Time::Time(int hour) : Time{hour, 0, 0} {}

// delegate to Time(int, int, int)
Time::Time(int hour, int minute) : Time{hour, minute, 0} {}
```

C++11 Delegating Constructors

In C++ 98

```
class X {
    int a;
    void validate(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
public:
    X(int x) { validate(x); }
    X() { validate(42); }
    X(string s) { int x = lexical_cast<int>(s); validate(x); }
    // ...
};
```

In C++11

```
class X {
    int a;
public:
    X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
    X() :X{42} { }
    X(string s) :X{lexical_cast<int>(s)} { }
    // ...
};
```

Destructors

A destructor is another type of special member function.

The name of the destructor for a class is the tilde character (~) followed by the class name (**Time::~Time()**).

Every class has exactly one destructor.

If you do NOT *explicitly* define a destructor, the compiler defines an “empty” destructor.

A class’s destructor is called *implicitly* when an object is destroyed.

The destructor itself does not actually release the object’s memory—it performs **termination housekeeping** before the object’s memory is reclaimed, so the memory may be reused to hold new objects.

Destructors are important for classes whose **objects contain dynamically allocated memory.**

When Constructors and Destructors Are Called

- Constructors and destructors are called **implicitly** when object are **created** and when they’re about to be **removed** from memory, respectively.

- The order in which these function calls occur 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**, but the **global** and **static** objects can alter the order in which destructors are called

When Constructors and Destructors Are Called

Constructors and Destructors for Objects in Global Scope:

- Constructors are called for objects defined in global scope **before any other function (including main) in that program begins execution** (although the order of execution of global object *constructors between files is not guaranteed*).
- The corresponding destructors are called when **main terminates**.

Constructors and Destructors for Non-static Local Objects:

- The constructor for a non-static local object is called when execution **reaches the point where that object is defined**—the corresponding destructor is called when **execution leaves the object's scope.**

Constructors and Destructors for static Local Objects:

- The constructor for a static local object is called only once, when execution first reaches the point where the object is defined—the corresponding destructor is called when main terminates

When Constructors and Destructors Are Called

```
class CreateAndDestroy {
public:
   CreateAndDestroy(int, std::string); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   sdt::string message; // message describing object
};

#endif
```

When Constructors and Destructors Are Called

```
#include <iostream>
#include "CreateAndDestroy.h"// include CreateAndDestroy class definition
using namespace std;

// constructor sets object's ID number and descriptive message
CreateAndDestroy::CreateAndDestroy(int ID, string messageString)
   : objectID{ID}, message{messageString} {
   cout << "Object " << objectID << "   constructor runs   "
      << message << endl;
}

// destructor
CreateAndDestroy::~CreateAndDestroy() {
   // output newline for certain objects; helps readability
   cout << (objectID == 1 || objectID == 6 ? "\n" : "");

   cout << "Object " << objectID << "   destructor runs    "
      << message << endl;
}
```

```cpp
#include <iostream>
#include "CreateAndDestroy.h" // include CreateAndDestroy class definition
using namespace std;

void create(); // prototype

CreateAndDestroy first{1, "(global before main)"}; // global object

int main() {
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second{2, "(local in main)"};
   static CreateAndDestroy third{3, "(local static in main)"};

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth{4, "(local in main)"};
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
}

// function to create objects
void create() {
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth{5, "(local in create)"};
   static CreateAndDestroy sixth{6, "(local static in create)"};
   CreateAndDestroy seventh{7, "(local in create)"};
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
}
```

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

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

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

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

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

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

When Constructors and Destructors Are Called

- Destructors are **not called for non-static** local objects if the program terminates with a call to function **exit** or function **abort.**

- Function **abort** performs similarly to function exit but forces the program to terminate immediately, **without allowing programmer-defined cleanup code of any kind to be called**. Destructors are **NOT** called.

When Constructors and Destructors Are Called

```
CreateAndDestroy first{1, "(global before main)"}; // global object

int main() {
  cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
  CreateAndDestroy second{2, "(local in main)"};
  static CreateAndDestroy third{3, "(local static in main)"};

  create(); // call function to create objects

  cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
  CreateAndDestroy fourth{4, "(local in main)"};
  exit(0);

  cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
}
```

When Constructors and Destructors Are Called

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

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

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

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

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local 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)
```

When Constructors and Destructors Are Called

```
CreateAndDestroy first{1, "(global before main)"}; // global object

int main() {
  cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
  CreateAndDestroy second{2, "(local in main)"};
  static CreateAndDestroy third{3, "(local static in main)"};

  create(); // call function to create objects

  cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
  CreateAndDestroy fourth{4, "(local in main)"};
  abort();

  cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
}
```

When Constructors and Destructors Are Called

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

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

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

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

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

Returning a reference to a private data member

A member function can return a reference to a private data member of that class.

If the reference return type is declared **const**, the reference is a nonmodifiable lvalue and cannot be used to modify the data.

However, if the reference return type is not declared const, subtle errors can occur.

- Returning a reference or a pointer to a private data member **breaks the encapsulation of the class and makes the client code dependent on the representation of the class's data**. However, there are cases where doing this is appropriate (we will see it later).

Returning a reference to a private data member

```
// prevent multiple inclusions of header
#ifndef TIME_H
#define TIME_H

class Time {
public:
   void setTime(int, int, int);
   unsigned int getHour() const;
   unsigned int& badSetHour(int); // dangerous reference return
private:
   unsigned int hour{0};
   unsigned int minute{0};
   unsigned int second{0};
};

#endif
```

```
#include <stdexcept>
#include "Time.h" // include definition of class Time
using namespace std;

// set values of hour, minute and second
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 hour value
unsigned int Time::getHour() const {return hour;}

// poor practice: returning a reference to a private data member.
unsigned int& Time::badSetHour(int hh) {
   if (hh >= 0 && hh < 24) {
      hour = hh;
   }
   else {
      throw invalid_argument("hour must be 0-23");
   }

   return hour; // dangerous reference return
}
```

```cpp
#include <iostream>
#include "Time.h" // include definition of class Time
using namespace std;

int main() {
   Time t; // create Time object

   // initialize hourRef with the reference returned by badSetHour
   unsigned int& hourRef{t.badSetHour(20)}; // 20 is a valid hour

   cout << "Valid hour before modification: " << hourRef;
   hourRef = 30; // use hourRef to set invalid value in Time object t
   cout << "\nInvalid hour after modification: " << t.getHour();

   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour(12) = 74; // assign another invalid value to hour

   cout << "\n\n*************************************************\n"
      << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
      << "t.badSetHour(12) as an lvalue, invalid hour: "
      << t.getHour()
      << "\n*************************************************" << endl;
}
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

*************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour(12) as an lvalue, invalid hour: 74
*************************************************
```

Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same class.

The compiler provides **default memberwise assignment operator** for *each class*.

By default, such assignment is performed by memberwise assignment (also called copy assignment)—each data member of the object on the *right* of the assignment operator is assigned individually to the same data member in the object on the *left* of the assignment operator.

Objects may be passed as function arguments and may be returned from functions.

In such cases, C++ creates a new object and uses a **copy constructor** to copy the original object's values into the new object. For each class, the compiler **provides a default copy constructor** that copies each member of the original object into the corresponding member of the new object.

Default Memberwise Assignment

```
#include <string>

// prevent multiple inclusions of header
#ifndef DATE_H
#define DATE_H

// class Date definition
class Date {
public:
   explicit Date(unsigned int = 1, unsigned int = 1, unsigned int = 2000);
   std::string toString() const;
private:
   unsigned int month;
   unsigned int day;
   unsigned int year;
};

#endif
```

Default Memberwise Assignment

```
#include <sstream>
#include <string>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

// Date constructor (should do range checking)
Date::Date(unsigned int m, unsigned int d, unsigned int y)
   : month{m}, day{d}, year{y} {}

// print Date in the format mm/dd/yyyy
string Date::toString() const {
   ostringstream output;
   output << month << '/' << day << '/' << year;
   return output.str();
}
```

Default Memberwise Assignment

```
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

int main() {
   Date date1{7, 4, 2004};
   Date date2; // date2 defaults to 1/1/2000

   cout << "date1 = " << date1.toString()
      << "\ndate2 = " << date2.toString() << "\n\n";

   date2 = date1; // default memberwise assignment

   cout << "After default memberwise assignment, date2 = "
      << date2.toString() << endl;
}
```

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

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

const Objects and const Member Functions

```
const Time noon{12, 0, 0};
Time currTime{11, 45, 0};
```

- *Attempts to **modify a const** object are caught at **compile time** rather than causing execution-time errors.*

- Declaring variables and objects **const** when appropriate <u>can improve performance</u>— compilers can perform *optimizations on constants* that cannot be performed on non-const variables.

- C++ **disallows** member-function calls for const objects unless the member functions themselves are **also declared const.**

- Defining as const a member function that calls a non-const member function or data member of the class on the same object is a compilation error.

- Invoking a <u>non-const member function on a const object</u> is a compilation error.

```cpp
#include "Time.h" // include Time class definition

int main() {
   Time wakeUp{6, 45, 0}; // non-constant object
   const Time noon{12, 0, 0}; // constant object

                              // OBJECT      MEMBER FUNCTION
   wakeUp.setHour(18);        // non-const   non-const
   noon.setHour(12);          // const       non-const
   wakeUp.getHour();          // non-const   const
   noon.getMinute();          // const       const
   noon.toUniversalString();  // const       const
   noon.toStandardString();   // const       non-const
}
```

Microsoft Visual C++ compiler error messages:

```
C:\examples\ch09\fig09_17\fig09_17.cpp(11): error C2662:
   'void Time::setHour(int)': cannot convert 'this' pointer from 'const Time'
   to 'Time &'
C:\examples\ch09\fig09_17\fig09_17.cpp(11): note: Conversion loses qualifiers
C:\examples\ch09\fig09_17\fig09_17.cpp(15): error C2662:
   'std::string Time::toStandardString(void)': cannot convert 'this' pointer
   from 'const Time' to 'Time &'
C:\examples\ch09\fig09_17\fig09_17.cpp(15): note: Conversion loses qualifiers
```

const Objects and const Member Functions

Constructors and Destructors cannot be const!

An interesting problem arises for constructors and destructors, each of which typically **modifies objects**.

- A constructor *must* be allowed to modify an object so that the object can be initialized.
- A destructor must be able to perform its termination housekeeping before an object's memory is reclaimed by the system.

❖ The "constness" of a const object is enforced from the time the constructor *completes* initialization of the object until that object's destructor is called.