

chapter

14

Methods

This chapter describes

- how constructors and destructors are defined to create and destroy objects
- how `inline` methods, access methods, and read-only methods can be used
- the pointer `this`, which is available for all methods, and
- what you need to pay attention to when passing objects as arguments or returning objects.

■ CONSTRUCTORS

Class Account with constructors

```
// account.h
// Defining class Account with two constructors.
// -----
#ifndef _ACCOUNT_
#define _ACCOUNT_
#include <string>
using namespace std;
class Account
{
private:                                // Sheltered members:
    string name;                        // Account holder
    unsigned long nr;                   // Account number
    double state;                       // State of the account
public:                                  // Public interface:
    Account( const string&, unsigned long, double );
    Account( const string& );
    bool init( const string&, unsigned long, double);
    void display();
};
#endif // _ACCOUNT_
```

Defining the constructors

```
// Within file account.cpp:

Account::Account( const string& a_name,
                  unsigned long a_nr, double a_state)
{
    nr    = a_nr;
    name  = a_name;
    state = a_state;
}

Account::Account( const string& a_name )
{
    name = a_name;
    nr = 1111111; state = 0.0;
}
```

□ The Task of a Constructor

Traditional programming languages only allocate memory for a variable being defined. The programmer must ensure that the variable is initialized with suitable values.

An object of the class `Account`, as described in the previous chapter, does not possess any valid values until the method `init()` is called. Non-initialized objects can lead to serious runtime errors in your programs.

To avoid errors of this type, C++ performs implicit initialization when an object is defined. This ensures that objects will always have valid data to work on. Initialization is performed by special methods known as *constructors*.

□ Declaration

Constructors can be identified by their names. In contrast to other member functions, the following applies:

- the name of the constructor is also the class name
- a constructor does not possess a return type—not even `void`.

Constructors are normally declared in the `public` section of a class. This allows you to create objects wherever the class definition is available.

Constructors can be overloaded, just like other functions. Constructors belonging to a class must be distinguishable by their *signature* (that is, the number, order, and type of parameters). This allows for different methods of object initialization. The example opposite shows an addition to the `Account` class. The class now has two constructors.

□ Definition

Since a constructor has the same name as its class, the definition of a constructor always begins with

```
Class_name::Class_name
```

In the definition itself, the arguments passed can be checked for validity before they are copied to the corresponding data members. If the number of arguments is smaller than the number of data members, the remaining members can be initialized using default values.

Constructors can also perform more complex initialization tasks, such as opening files, allocating memory, and configuring interfaces.

■ CONSTRUCTOR CALLS

Sample program

```
// account2_t.cpp
// Using the constructors of class Account.
// -----

#include "account.h"

int main()
{
    Account giro("Cheers, Mary", 1234567, -1200.99 ),
           save("Lucky, Luke");

    Account depot; // Error: no default constructor
                  //         defined.

    giro.display(); // To output
    save.display();

    Account temp("Funny, Susy", 7777777, 1000000.0);
    save = temp; // ok: Assignment of
                //         objects possible.

    save.display();

    // Or by the presently available method init():
    save.init("Lucky, Luke", 7654321, 1000000.0);
    save.display();

    return 0;
}
```

Unlike other methods, constructors cannot be called for existing objects. For this reason, a constructor does not have a return type. Instead, a suitable constructor is called once only when an object is created.

□ Initialization

When an object is defined, initial values can follow the object name in parentheses.

Syntax: `class object(initializing_list);`

During initialization the compiler looks for a constructor whose signature matches the initialization list. After allocating sufficient memory for the object, the constructor is called. The values in the initialization list are passed as arguments to the constructor.

Example: `account nomoney("Poor, Charles");`

This statement calls the constructor with one parameter for the name. The other data members will default to standard values.

If the compiler is unable to locate a constructor with a suitable signature, it will not create the object but issue an error message.

Example: `account somemoney("Li, Ed",10.0); // Error!`

The class `Account` does not contain a constructor with two parameters.

If a constructor with only *one* parameter is defined in the class, the statement can be written with an equals sign `=`.

Example: `account nomoney = "Poor, Charles";`

This statement is equivalent to the definition in the example before last. Initialization with parentheses or the `=` sign was introduced previously for fundamental types. For example, `int i(0);` is equivalent to `int i=0;`

□ Default Constructor

A constructor *without* parameters is referred to as a *default constructor*. The default constructor is only called if an object definition does not explicitly initialize the object. A default constructor will use standard values for all data members.

If a class does not contain a constructor definition, the compiler will create a minimal version of the default constructor as a `public` member. However, this constructor will not perform initialization. By contrast, if a class contains *at least one* constructor, a default constructor must be defined explicitly, if it is needed. The definition of the `Account` class does not specify a default constructor; thus a new account object can be created with initialization only.

■ DESTRUCTORS

Sample program

```
// demo.cpp
// Outputs constructor and destructor calls.
// -----
#include <iostream>
#include <string>
using namespace std;
int count = 0; // Number of objects.
class Demo
{
private:   string name;
public:   Demo( const string& ); // Constructor
         ~Demo(); // Destructor
};
Demo::Demo( const string& str)
{
    ++count; name = str;
    cout << "I am the constructor of "<< name << '\n'
         << "This is the " << count << ". object!\n"
}
Demo:: ~Demo() // Defining the destructor
{
    cout << "I am the destructor of " << name << '\n'
         << "The " << count << ". object "
         << "will be destroyed " << endl;
    --count;
}
// -- To initialize and destroy objects of class Demo --
Demo globalObject("the global object");
int main()
{
    cout << "The first statement in main()." << endl;
    Demo firstLocalObject("the 1. local object");
    {
        Demo secLocalObject("the 2. local object");
        static Demo staticObject("the static object");
        cout << "\nLast statement within the inner block"
             << endl;
    }
    cout << "Last statement in main()." << endl;
    return 0;
}
```

□ Cleaning Up Objects

Objects that were created by a constructor must also be cleaned up in an orderly manner. The tasks involved in cleaning up include releasing memory and closing files.

Objects are cleaned up by a special method called a *destructor*, whose name is made up of the class name preceded by ~ (tilde).

□ Declaration and Definition

Destructors are declared in the `public` section and follow this syntax:

Syntax: `~class_name(void);`

Just like the constructor, a destructor does not have a return type. Neither does it have any parameters, which makes the destructor impossible to overload. Each class thus has *one* destructor only.

If the class does not define a destructor, the compiler will create a minimal version of a destructor as a `public` member, called the *default destructor*.

It is important to define a destructor if certain actions performed by the constructor need to be undone. If the constructor opened a file, for example, the destructor should close that file. The destructor in the `Account` class has no specific tasks to perform. The explicit definition is thus:

```
Account::~Account() {}           // Nothing to do
```

The individual data members of an object are always removed in the order opposite of the order in which they were created. The first data member to be created is therefore cleaned up last. If a data member is also a class type object, the object's own destructor will be called.

□ Calling Destructors

A destructor is called automatically at the end of an object's lifetime:

- for local objects except objects that belong to the `static` storage class, at the end of the code block defining the object
- for global or `static` objects, at the end of the program.

The sample program on the opposite page illustrates various implicit calls to constructors and destructors.

■ INLINE METHODS

Sample class Account

```

// account.h
// New definition of class Account with inline methods
// -----
#ifndef _ACCOUNT_
#define _ACCOUNT_

#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

class Account
{
private:          // Sheltered members:
    string name;          // Account holder
    unsigned long nr;     // Account number
    double state;        // State of the account
public:           // Public interface:
    // Constructors: implicit inline
    Account( const string& a_name = "X",
              unsigned long a_nr   = 1111111L,
              double a_state      = 0.0)
    {
        name = a_name; nr = a_nr; state = a_state;
    }
    ~Account(){ } // Dummy destructor: implicit inline
    void display();
};

// display() outputs data of class Account.
inline void Account::display() // Explicit inline
{
    cout << fixed << setprecision(2)
         << "-----\n"
         << "Account holder:  " << name << '\n'
         << "Account number:  " << nr << '\n'
         << "Account state:   " << state << '\n'
         << "-----\n"
         << endl;
}
#endif // _ACCOUNT_

```

A class typically contains multiple methods that fulfill simple tasks, such as reading or updating data members. This is the only way to ensure data encapsulation and class functionality.

However, continually calling “short” methods can impact a program’s runtime. In fact, saving a re-entry address and jumping to the called function and back into the calling function can take more time than executing the function itself. To avoid this overhead, you can define `inline` methods in a way similar to defining `inline` global functions.

□ Explicit and Implicit `inline` Methods

Methods can be explicitly or implicitly defined as `inline`. In the first case, the method is declared within the class, just like any other method. You simply need to place the `inline` keyword before the method name in the function header when defining the method.

```
Example: inline void Account::display()
{
    . . .
}
```

Since the compiler must have access to the code block of an `inline` function, the `inline` function should be defined in the header containing the class definition.

Short methods can be defined within the class. Methods of this type are known as implicit `inline` methods, although the `inline` keyword is not used.

```
Example: // Within class Account:
bool isPositive(){ return state > 0; }
```

□ Constructors and Destructors with `inline` Definitions

Constructors and destructors are special methods belonging to a class and, as such, can be defined as `inline`. This point is illustrated by the new definition of the `Account` class opposite. The constructor and the destructor are both implicit `inline`. The constructor has a default value for each argument, which means that we also have a default constructor. You can now define objects without supplying an initialization list.

```
Example: Account temp;
```

Although we did not explicitly supply values here, the object `temp` was correctly initialized by the default constructor we defined.

■ ACCESS METHODS

Access methods for the Account class

```

// account.h
// Class Account with set- and get-methods.
// -----
#ifndef _ACCOUNT_
#define _ACCOUNT_

#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

class Account
{
private:           // Sheltered members:
    string name;           // Account holder
    unsigned long nr;     // Account number
    double state;        // State of the account
public:           // Public interface:
    Account( const string& a_name = "X",
             unsigned long a_nr  = 1111111L,
             double a_state     = 0.0)
    { name = a_name; nr = a_nr; state = a_state; }
    ~Account() { }

    // Access methods:
    const string& getName() { return name; }
    bool setName( const string& s)
    {
        if( s.size() < 1) // No empty name
            return false;
        name = s;
        return true;
    }
    unsigned long getNr() { return nr; }
    void setNr( unsigned long n) { nr = n; }
    double getState() { return state; }
    void setState(double x) { state = x; }
    void display();
};
// inline definition of display() as before.
#endif // _ACCOUNT_

```

□ Accessing Private Data Members

An object's data members are normally found in the `private` section of a class. To allow access to this data, you could place the data members in the `public` section of the class; however, this would undermine any attempt at data encapsulation.

Access methods offer a far more useful way of accessing the private data members. Access methods allow data to be read and manipulated in a controlled manner. If the access methods were defined as `inline`, access is just as efficient as direct access to the `public` members.

In the example opposite, several access methods have been added to the `Account` class. You can now use the

```
getName(), getNr(), getState()
```

methods to read the individual data members. As is illustrated in `getName()`, references should be read-only when used as return values. Direct access for write operations could be possible otherwise. To manipulate data members, the following methods can be used:

```
setName(), setNr(), setState().
```

This allows you to define a new balance, as follows:

Example: `save.setState(2199.0);`

□ Access Method Benefits

Defining access methods for reading and writing to each data member may seem like a lot of work—all that typing, reams of source code, and the programmer has to remember the names and tasks performed by all those methods.

So, you may be asking yourself how you benefit from using access methods. There are two important issues:

- Access methods can prevent invalid access attempts at the onset by performing sanity checks. If a class contains a member designed to represent positive numbers only, an access method can prevent processing negative numbers.
- Access methods also hide the actual implementation of a class. It is therefore possible to modify the internal structure of your data at a later stage. If you detect that a new data structure will allow more efficient data handling, you can add this modification to a new version of the class. Provided the public interface to the class remains unchanged, an application program can be leveraged by the modification without needing to modify the application itself. You simply recompile the application program.

■ const OBJECTS AND METHODS

Read-only methods in the Account class

```

// account.h
// Account class with read-only methods.
// -----
#ifndef _ACCOUNT_
#define _ACCOUNT_

#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

class Account
{
private:          // Sheltered members
  // Data members: as before
public:          // Public interface
  // Constructors and destructor
  . . . // as before

  // Get-methods:
  const string& getName() const { return name; }
  unsigned long getNr() const { return nr; }
  double      getState() const { return state; }
  // Set-methods:
  . . . // as before
  // Additional methods:
  void display() const;
};
// display() outputs the data of class Account.
inline void Account::display() const
{
  cout << fixed << setprecision(2)
    << "-----\n"
    << "Account holder:  " << name << '\n'
    << "Account number:  " << nr << '\n'
    << "Account state:   " << state << '\n'
    << "-----\n"
    << endl;
}
#endif // _ACCOUNT_

```

□ Accessing `const` Objects

If you define an object as `const`, the program can only read the object. As mentioned earlier, the object must be initialized when you define it for this reason.

Example: `const Account inv("YMCA, FL", 5555, 5000.0);`

The object `inv` cannot be modified at a later stage. This also means that methods such as `setName()` cannot be called for this object. However, methods such as `getName` or `display()` will be similarly unavailable although they only perform read access with the data members.

The reason for this is that the compiler cannot decide whether a method performs write operations or only read operations with data members unless additional information is supplied.

□ Read-Only Methods

Methods that perform only read operations and that you need to call for constant objects must be identified as read-only. To identify a method as read-only, append the `const` keyword in the method declaration and in the function header for the method definition.

Example: `unsigned long getNr() const;`

This declares the `getNr()` method as a *read-only method* that can be used for constant objects.

Example: `cout << "Account number: " << inv.getNr();`

Of course, this does not prevent you from calling a read-only method for a non-constant object.

The compiler issues an error message if a read-only method tries to modify a data member. This also occurs when a read-only method calls another method that is not defined as `const`.

□ `const` and Non-`const` Versions of a Method

Since the `const` keyword is part of the method's signature, you can define two versions of the method: a read-only version, which will be called for constant objects by default, and a normal version, which will be called for non-`const` objects.

■ STANDARD METHODS

Sample program

```

// stdMeth.cpp
// Using standard methods.
// -----
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

class CD
{ private:
  string interpret, title;
  long  seconds;          // Time duration of a song
public:
  CD( const string& i="", const string& t="", long s = 0L)
  {
    interpret = i;    title = t;    seconds = s;
  }
  const string& getInterpret() const{ return interpret; }
  const string& getTitle() const   { return title; }
  long getSeconds() const          { return seconds; }
};
// Generate objects of class CD and output it in tabular form
void printLine( CD cd) ;          // A row of the table
int main()
{
  CD cd1( "Mister X", "Let's dance", 30*60 + 41),
    cd2( "New Guitars", "Flamenco Collection", 2772 ),
    cd3 = cd1,                    // Copy constructor!
    cd4;                          // Default constructor.
    cd4 = cd2;                    // Assignment!
  string line( 70, '-');  line += '\n';
  cout << line << left
        << setw(20) << "Interpreter" << setw(30) << "Title"
        << "Length (Min:Sec)\n" << line << endl;
  printLine(cd3);                 // Call by value ==>
  printLine(cd4);                 // Copy constructor!
  return 0;
}
void printLine( CD cd)
{  cout << left << setw(20) << cd.getInterpret()
    << setw(30) << cd.getTitle()
    << right << setw(5) << cd.getSeconds() / 60
    << ':' << setw(2) << cd.getSeconds() % 60 << endl;
}

```

Every class *automatically* contains four standard methods:

- the default constructor
- the destructor
- the copy constructor and
- the assignment.

You can use your own definitions to replace these standard methods. As illustrated by the sample class `Account`, the compiler only uses the pre-defined default constructor if no other constructor is available.

The default constructor and the implicit, minimal version of a destructor were introduced earlier.

□ Copy Constructor

The copy constructor initializes an object with another object of the same type. It is called automatically when a second, already existing object is used to initialize an object.

Example:

```
Account myAccount("Li, Ed", 2345, 124.80);
Account yourAccount(myAccount);
```

In this example, the object `yourAccount` is initialized by calling the copy constructor with the `myAccount` object. Each member is copied individually, that is, the following initialization process takes place:

```
yourAccount.name = myAccount.name;
yourAccount.nr   = myAccount.nr;
yourAccount.state = myAccount.state;
```

The copy constructor is also called when an object is passed to a function by value. When the function is called, the parameter is created and initialized with the object used as an argument.

□ Assignment

Assignment has been used in several previous examples. An object can be assigned to another object of the same type.

Example:

```
hisAccount = yourAccount;
```

The data members of the `yourAccount` object are copied to the corresponding members of `hisAccount` in this case also. In contrast to initialization using the copy constructor, assignment requires two *existing* objects.

Later in the book, you will be introduced to situations where you need to define the copy constructor or an assignment yourself, and the necessary techniques will be discussed.

■ **this** POINTERSample class `DayTime`

```

// DayTime.h
// The class DayTime represents the time in
// hours, minutes and seconds.
// -----
#ifndef _DAYTIME_
#define _DAYTIME_
class DayTime
{
private:
    short hour, minute, second;
    bool overflow;
public:
    DayTime( int h = 0, int m = 0, int s = 0)
    {
        overflow = false;
        if( !setTime( h, m, s) ) // this->setTime(...)
            hour = minute = second = 0; // hour is equivalent
        // to this->hour etc.
    }
    bool setTime(int hour, int minute, int second = 0)
    {
        if(    hour    >= 0    &&    hour    < 24
            &&    minute >= 0    &&    minute < 60
            &&    second >= 0    &&    second < 60 )
        {
            this->hour   = (short)hour;
            this->minute = (short)minute;
            this->second = (short)second;
            return true;
        }
        else
            return false;
    }
    int getHour()    const { return hour;    }
    int getMinute() const { return minute;  }
    int getSecond() const { return second;  }

    int asSeconds() const // daytime in seconds
    { return (60*60*hour + 60*minute + second); }

    bool isLess( DayTime t) const // compare *this and t
    {
        return asSeconds() < t.asSeconds();
    } // this->asSeconds() < t.asSeconds();
};
#endif // _DAYTIME_

```

□ Accessing the Current Object

A method can access any member of an object without the object name being supplied in every case. A method will always reference the object with which it was called.

But how does a method know which object it is currently working with? When a method is called, it is passed a hidden argument containing the address of the current object.

The address of the current object is available to the method via the constant pointer `this`. Given that `actObj` is the current object of type `Class_id`, for which a method was called, the pointer `this` has the following declaration:

```
Class_id* const this = &actObj;
```

The name `this` is a keyword. As `this` is a constant pointer, it cannot be redirected. In other words, the pointer `this` allows you to access the current object only.

□ Using the `this` Pointer

You can use the `this` pointer within a method to address an object member as follows:

```
Example:  this->data    // Data member: data
           this->func()  // Calling member function
```

The compiler implicitly creates an expression of this type if only a member of the current object is supplied.

```
Example: data = 12; // Corresponds to this->data=12;
```

Write operations of this type are permissible since the pointer `this` is a constant, but the referenced object is not. However, the above statement would be invalid for a read-only method.

The `this` pointer can be used explicitly to distinguish a method's local variables from class members of the same name. This point is illustrated by the sample method `setTime()` on the opposite page.

The `this` pointer is always necessary to access the current object, `*this`, collectively. This situation often occurs when the current object needs to be returned as a copy or by reference. Then the return statement is as follows:

```
return *this;
```

■ PASSING OBJECTS AS ARGUMENTS

Calling methods `setTime()` and `isLess()`

```
#include "DayTime.h"
. . .
DayTime depart1( 11, 11, 11), depart2;
. . .
depart2.setTime(12, 0, 0);
if( depart1.isLess( depart2) )
    cout << "\nThe 1st plane takes off earlier" << endl;
. . .
```

Global function `swap()`

```
#include "DayTime.h"
// Defines the global function swap():
void swap( DayTime& t1, DayTime& t2)      // Two
{                                         // parameters!
    DayTime temp(t1); t1 = t2; t2 = temp; // To swap
}                                         // t1 and t2.
// A call (e.g. in function main()):
DayTime arrival1( 14, 10), arrival2( 15, 20);
. . .
swap( arrival1, arrival2);              // To swap
. . .
```

Implementing `swap()` as a method

```
// Defines the method swap():
class DayTime                          // With a new method swap()
{ . . .
public:
    void swap( DayTime& t)              // One parameter!
    {                                   // To swap *this and t:
        DayTime temp(t); t = *this; *this = temp;
    }
};
// A call (e.g. in function main()):
#include "DayTime.h"
DayTime arrival1( 10, 10), arrival2( 9, 50);
. . .
arrival1.swap(arrival2);
. . .
```

□ Passing by Value

As you already know, passing by value copies an object that was passed as an argument to the corresponding parameter of a function being called. The parameter is declared as an object of the class in question.

Example: `bool isLess(DayTime t) const;`

When the method `isLess()` is called, the copy constructor executes and initializes the created object, `t`, with the argument.

```
depart1.isLess( depart2)    // Copy constructor
```

The function uses a copy of the object `depart2`. The copy is cleaned up when leaving the function, that is, the destructor is called.

□ Passing by Reference

The overhead caused by creating and cleaning up objects can be avoided by passing arguments by reference. In this case, the parameter is declared as a reference or pointer.

Example: `bool isLess(const DayTime& t) const;`

This new declaration of the `isLess()` method is preferable to the previous declaration. There is no formal difference to the way the method is called. However, `isLess()` no longer creates an internal copy, but accesses directly the object being passed. Of course, the object cannot be changed, as the parameter was declared read-only.

□ Methods Versus Global Functions

Of course, it is possible to write a global function that expects *one* object as an argument. However, this rarely makes sense since you would normally expect an object's functionality to be defined in the class itself. Instead, you would normally define a method for the class and the method would perform the task in hand. In this case, the object would not be passed as an argument since the method would manipulate the members of the current object.

A different situation occurs where operations with at least *two* objects need to be performed, such as comparing or swapping. For example, the method `isLess()` could be defined as a global function with two parameters. However, the function could only access the public interface of the objects. The function `swap()` on the opposite page additionally illustrates this point.

The major advantage of a globally defined function is its symmetry. The objects involved are peers, since both are passed as arguments. This means that conversion rules are applied to both arguments when the function is called.

■ RETURNING OBJECTS

Global function `currentTime()`

```
#include "DayTime.h"
#include <ctime>          // Functions time(), localtime()
using namespace std;

const DayTime& currentTime()    // Returns the
{                               // present time.
    static DayTime curTime;
    time_t sec; time(&sec); // Gets the present time.
                          // Initializes the struct
    struct tm *time = localtime(&sec); // tm with it.

    curTime.setTime( time->tm_hour, time->tm_min,
                    time->tm_sec );

    return curTime;
}
```

Sample program

```
// DayTim_t.cpp
// Tests class DayTime and function currentTime()
// -----
#include "DayTime.h"          // Class definition
#include <iostream>
using namespace std;

const DayTime& currentTime(); // The current time.

int main()
{
    DayTime cinema( 20,30);

    cout << "\nThe movie starts at ";
    cinema.print();

    DayTime now(currentTime()); // Copy constructor
    cout << "\nThe current time is ";
    now.print();

    cout << "\nThe movie has ";
    if( cinema.isLess( now) )
        cout << "already begun!\n" << endl;
    else
        cout << "not yet begun!\n" << endl;
    return 0;
}
```

A function can use the following ways to return an object as a return value: It can create a copy of the object, or it can return a reference or pointer to that object.

□ Returning a Copy

Returning a copy of an object is time-consuming and only makes sense for small-scale objects.

Example:

```
DayTime startMeeting()
{
    DayTime start;
    . . . // Everyone has time at 14:30:
    start.setTime( 14, 30);
    return( start);
}
```

On exiting the function, the local object `start` is destroyed. This forces the compiler to create a temporary copy of the local object and return the copy to the calling function.

□ Returning a Reference

Of course, it is more efficient to return a reference to an object. But be aware that the lifetime of the referenced object must not be local.

If this is the case, the object is destroyed on exiting the function and the returned reference becomes invalid. If you define the object within a function, you must use a `static` declaration.

The global function `currentTime()` on the opposite page exploits this option by returning a reference to the current time that it reads from the system each time the function is called. The sample program that follows this example uses the current time to initialize the new object `now` and then outputs the time. In order to output the time, an additional method, `print()`, was added to the class.

□ Using Pointers as Return Values

Instead of returning a reference, a function can also return a pointer to an object. In this case too, you must ensure that the object still exists after exiting the function.

Example:

```
const DayTime* currentTime() // Read-only pointer
{
    . . . // Unchanged // to the current time
    return &curTime;
}
```

exercises

■ EXERCISES

Class Article

Private members:

	Type
Article number:	long
Article name:	string
Sales price:	double

Public members:

```
Article(long, const string&, double);  
~Article();  
void print(); // Formatted output  
set- and get-methods for any data member
```

Output from constructor

```
An object of type Article . . . is created.  
This is the . . . . Article.
```

Output from destructor

```
The object of type Article . . . is destroyed.  
There are still . . . articles.
```

Exercise I

A warehouse management program needs a class to represent the articles in stock.

- Define a class called `Article` for this purpose using the data members and methods shown opposite. Store the class definition for `Article` in a separate header file. Declare the constructor with default arguments for each parameter to ensure that a default constructor exists for the class. Access methods for the data members are to be defined as `inline`. Negative prices must not exist. If a negative price is passed as an argument, the price must be stored as `0.0`.
- Implement the constructor, the destructor, and the method `print()` in a separate source file. Also define a global variable for the number of `Article` type objects.

The constructor must use the arguments passed to it to initialize the data members, additionally increment the global counter, and issue the message shown opposite.

The destructor also issues a message and decrements the global counter.

The method `print()` displays a formatted object on screen. After outputting an article, the program waits for the return key to be pressed.

- The application program (again use a separate source file) tests the `Article` class. Define four objects belonging to the `Article` class type:
 1. A global object and a local object in the `main` function.
 2. Two local objects in a function `test()` that is called twice by `main()`. One object needs a static definition. The function `test()` displays these objects and outputs a message when it is terminated.

Use articles of your own choice to initialize the objects. Additionally, call the access methods to modify individual data members and display the objects on screen.

- Test your program. Note the order in which constructors and destructors are called.

Supplementary question: Suppose you modify the program by declaring a function called `test()` with a parameter of type `Article` and calling the function with an article type object. The counter for the number of objects is negative after running the program. Why?

Methods for class `Date`

Public Methods:

```
Date();
Date( int month, int day, int year);
void setDate();
bool setDate( int mn, int da, int yr);

int getMonth() const;
int getDay() const;
int getYear() const;
bool isEqual( const Date&) const;
bool isLess( const Date&) const;
const string& asString() const;
void print() const;
```

Converting a number to a string

The class `stringstream` offers the same functionality for reading and writing to character buffer as the classes `istream` and `ostream` do. Thus, the operators `>>` and `<<`, just as all manipulators, are available.

```
// Example: Converting a number to a string.
#include <sstream>           // Class stringstream
#include <iomanip>           // Manipulators

double x = 12.3;           // Number
string str;                // Destination string
stringstream iostream;    // For conversion
                           // number -> string.
iostream << setw(10) << x; // Add to the stream.
iostream >> str;           // Read from the stream.
```

Notices for exercise 3

- A year is a leap year if it is divisible by 4 but not by 100. In addition, all multiples of 400 are leap years. February has 29 days in a leap year.
- Use a `switch` statement to examine the number of days for months containing less than 31 days.

Exercise 2

In the exercises for chapter 13, an initial version of the `Date` class containing members for day, month, and year was defined. Now extend this class to add additional functionality. The methods are shown on the opposite page.

- The constructors and the method `setDate()` replace the `init` method used in the former version. The default constructor uses default values, for example, `1.1.1`, to initialize the objects in question. The `setDate()` method without any parameters writes the current date to the object.
- The constructor and the `setDate()` method with three parameters do not need to perform range checking. This functionality will be added in the next exercise.
- The methods `isEqual()` and `isLess()` enable comparisons with a date passed to them.
- The method `asString()` returns a reference to a string containing the date in the format `mm-dd-year`, e.g. `03-19-2006`. You will therefore need to convert any numerical values into their corresponding decimal strings. This operation is performed automatically when you use the `<<` operator to output a number to the standard output `cout`. In addition to the `cin` and `cout` streams, with which you are already familiar, so-called *string streams* with the same functionality also exist. However, a string stream does not read keyboard input or output data on screen. Instead, the target, or source, is a buffer in main memory. This allows you to perform formatting and conversion in main memory.
- Use an application program that calls all the methods defined in the class to test the `Date` class.

Exercise 3

The `Date` class does not ensure that an object represents a valid date. To avoid this issue, add range checking functionality to the class. Range checking is performed by the constructor and the `setDate()` method with three parameters.

- First, write a function called `isLeapYear()` that belongs to the `bool` type and checks whether the year passed to it is a leap year. Define the function as a global `inline` function and store it in the header file `Date.h`.
- Modify the `setDate()` method to allow range checking for the date passed to it. The constructor can call `setDate()`.
- Test the new version of the `Date` class. To do so, and to test `setDate(...)`, read a date from the keyboard.

■ SOLUTIONS

Exercise 1

```

// -----
// article.h
// Defines a simple class, Article.
// -----
#ifdef _ARTICLE_
#define _ARTICLE_

#include <string>
using namespace std;

class Article
{
private:
    long nr;                // Article number
    string name;           // Article name
    double sp;             // Selling price

public:
    Article( long nr=0, const string& name="noname",
            double sp=0.0);
    ~Article();
    void print();

    const string& getName() const { return name; }
    long         getNr()   const { return nr; }
    double       getSP()   const { return sp; }

    bool setName( const string& s)
    {
        if( s.size() < 1)           // No empty name
            return false;
        name = s;
        return true;
    }
    void setNr( long n) { nr = n; }
    void setSP(double v)
    {
        sp = v > 0.0 ? v : 0.0;    // No negative price
    }
};

#endif // _ARTICLE_

```

```

// -----
// article.cpp
// Defines those methods of Article, which are
// not defined inline.
// Screen output for constructor and destructor calls.
// -----
#include "Article.h"          // Definition of the class
#include <iostream>
#include <iomanip>
using namespace std;

// Global counter for the objects:
int count = 0;

// -----
// Define constructor and destructor:
Article::Article( long nr, const string& name, double sp)
{
    setNr(nr);   setName(name);   setSP(sp);
    ++count;
    cout << "Created object for the article " << name
         << ".\n"
         << "This is the " << count << ". article!\n"
    }
Article::~Article()
{
    cout << "Cleaned up object for the article " << name
         << ".\n"
         << "There are still " << --count << " articles!"
         << endl;
}
// -----
// The method print() outputs an article.
void Article::print()
{
    long savedFlags = cout.flags();          // To mark the
                                           // flags of cout.

    cout << fixed << setprecision(2)
         << "-----\n"
         << "Article data:\n"
         << "  Number ....: " << nr   << '\n'
         << "  Name ....: " << name << '\n'
         << "  Sales price: " << sp  << '\n'
         << "-----"
         << endl;
    cout.flags(savedFlags);                 // To restore
                                           // old flags.

    cout << "  --- Go on with return --- ";
    cin.get();
}

```

```
// -----  
// article_t.cpp  
// Tests the Article class.  
// -----  
#include "Article.h"      // Definition of the class  
#include <iostream>  
#include <string>  
using namespace std;  
  
void test();  
  
// -- Creates and destroys objects of Article class --  
Article Article1( 1111,"volley ball", 59.9);  
int main()  
{  
    cout << "\nThe first statement in main().\n" << endl;  
    Article Article2( 2222,"gym-shoes", 199.99);  
    Article1.print();  
    Article2.print();  
    Article& shoes = Article2;          // Another name  
    shoes.setNr( 2233);  
    shoes.setName("jogging-shoes");  
    shoes.setSP( shoes.getSP() - 50.0);  
  
    cout << "\nThe new values of the shoes object:\n";  
    shoes.print();  
    cout << "\nThe first call to test()." << endl;  
    test();  
    cout << "\nThe second call to test()." << endl;  
    test();  
    cout << "\nThe last statement in main().\n" << endl;  
    return 0;  
}  
  
void test()  
{  
    Article shirt( 3333, "T-Shirt", 29.9);  
    shirt.print();  
    static Article net( 4444, "volley ball net", 99.0);  
    net.print();  
    cout << "\nLast statement in function test()" << endl;  
}
```

Answer to the supplementary question:

The copy constructor is called on each “passing by value,” although this constructor has not been defined explicitly. In other words, the implicitly defined copy constructor is used and of course does not increment the object counter. However, the explicitly defined destructor, which decrements the counter, is still called for each object.

Exercises 2 and 3

```
// -----  
// Date.h  
// Defining class Date with optimized  
// functionality, e.g. range check.  
// -----  
#ifndef _DATE_ // Avoids multiple inclusions.  
#define _DATE_  
#include <string>  
using namespace std;  
  
class Date  
{  
private:  
    short month, day, year;  
public:  
    Date() // Default constructor  
    { month = day = year = 1; }  
  
    Date( int month, int day, int year)  
    {  
        if( !setDate( month, day, year) )  
            month = day = year = 1; // If date is invalid  
    }  
    void setDate(); // Sets the current date  
    bool setDate( int month, int day, int year);  
    int getMonth() const { return month; }  
    int getDay() const { return day; }  
    int getYear() const { return year; }  
    bool isEqual( const Date& d) const  
    {  
        return month == d.month && day == d.day  
            && year == d.year ;  
    }  
    bool isLess( const Date& d) const;  
    const string& asString() const;  
    void print(void) const;  
};
```

```
inline bool Date::isLess( const Date& d) const
{
    if( year != d.year)          return year < d.year;
    else if( month != d.month)  return month < d.month;
    else                          return day < d.day;
}

inline bool isLeapYear( int year)
{
    return (year%4 == 0 && year%100 != 0) || year%400 == 0;
}
#endif // _DATE_

// -----
// Date.cpp
// Implements those methods of Date class,
// which are not defined inline.
// -----
#include "Date.h" // Class definition
#include <iostream>
#include <sstream>
#include <iomanip>
#include <string>
#include <ctime>
using namespace std;

// -----
void Date::setDate() // Get the present date and
{ // assign it to the data members.
    struct tm *dur; // Pointer to struct tm.
    time_t sec; // For seconds.

    time(&sec); // Get the present time.
    dur = localtime(&sec); // Initialize a struct of
    // type tm and return a
    // pointer to it.
    day = (short) dur->tm_mday;
    month = (short) dur->tm_mon + 1;
    year = (short) dur->tm_year + 1900;
}
```

```
// -----
bool Date::setDate( int mn, int da, int yr)
{
    if( mn < 1 || mn > 12 ) return false;
    if( da < 1 || da > 31 ) return false;

    switch(mn)           // Month with less than 31 days
    {
        case 2:  if( isLeapYear(yr))
                  {
                      if( da > 29)
                          return false;
                  }
                else if( da > 28)
                    return false;
                break;

        case 4:
        case 6:
        case 9:
        case 11:
                if( da > 30) return false;
    }
    month = (short) mn;
    day   = (short) da;
    year  = (short) yr;
    return true;
}

// -----
void Date::print() const           // Output a date
{
    cout << asString() << endl;
}

// -----
const string& Date::asString() const // Return a date
{                                     // as string.
    static string dateString;
    stringstream iostream;           // For conversion
                                        // number -> string
    iostream << setfill('0')           // and formatting.
                << setw(2) << month << '-'
                << setw(2) << day  << '-' << year;

    iostream >> dateString;
    return dateString;
}
```

```
// -----  
// date_t.cpp  
// Using objects of class Date.  
// -----  
#include "Date.h"  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    Date today, birthday( 1, 29, 1927);  
    const Date d2010(1,1,2010);  
  
    cout << "\n Brigit's birthday: "  
         << birthday.asString() << endl;  
  
    today.setDate();  
    cout << "\nToday's date: " << today.asString()  
         << endl;;  
  
    if( today.isLess( d2010))  
        cout << " Good luck for this decade \n"  
             << endl;  
    else  
        cout << " See you next decade \n" << endl;  
  
    Date holiday;  
    int month, day, year;    char c;  
  
    cout << "\nWhen does your next vacation begin?\n"  
         << "Enter in Month-Day-Year format: ";  
  
    if( !(cin >> month >> c >> day >> c >> year) )  
        cerr << "Invalid input!\n" << endl;  
    else if ( !holiday.setDate( month, day, year))  
        cerr << "Invalid date!\n" << endl;  
    else  
    {  
        cout << "\nYour first vacation: ";  
        holiday.print();  
  
        if( today.getYear() < holiday.getYear())  
            cout << "You should go on vacation this year!\n"  
                 << endl;  
        else  
            cout << "Have a nice trip!\n" << endl;  
    }  
    return 0;  
}
```