Illustrations List *(Main Page)*

```
// Fig. 7.1: time5.h
// Declaration of the class Time.
// Member functions defined in time5.cpp
#ifndef TIME5_H
#define TIME5_H

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

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

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

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard();        // print standard time
```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 1 of 6).

```
private:
   int hour;              // 0 - 23
   int minute;            // 0 - 59
   int second;            // 0 - 59
};

#endif
```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 2 of 6).

```
// Fig. 7.1: time5.cpp
// Member function definitions for Time class.
#include <iostream.h>
#include "time5.h"

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );
}

// Set the hour value
void Time::setHour( int h )
   { hour = ( h >= 0 && h < 24 ) ? h : 0; }

// Set the minute value
void Time::setMinute( int m )
   { minute = ( m >= 0 && m < 60 ) ? m : 0; }

```

```
// Set the second value
void Time::setSecond( int s )
   { second = ( s >= 0 && s < 60 ) ? s : 0; }

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

// Get the minute value
int Time::getMinute() const { return minute; }

// Get the second value
int Time::getSecond() const { return second; }
```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 3 of 6).

```

// Display military format time: HH:MM
void Time::printMilitary() const
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard()
{
   cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
        << ( minute < 10 ? "0" : "" ) << minute << ":"
        << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 4 of 6).

```
// Fig. 7.1: fig07_01.cpp
// Attempting to access a const object with
// non-const member functions.
#include <iostream.h>
#include "time5.h"

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

                          // MEMBER FUNCTION   OBJECT
   wakeUp.setHour( 18 );  // non-const         non-const

   noon.setHour( 12 );    // non-const         const

   wakeUp.getHour();      // const             non-const

   noon.getMinute();      // const             const
   noon.printMilitary();  // const             const
   noon.printStandard();  // non-const         const
   return 0;
}
```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 5 of 6).

```
Compiling Fig07_01.cpp
Fig07_01.cpp(15) : error: 'setHour' :
   cannot convert 'this' pointer from
   'const class Time' to 'class Time &'
   Conversion loses qualifiers
Fig07_01.cpp(21) : error: 'printStandard' :
   cannot convert 'this' pointer from
   'const class Time' to 'class Time &'
   Conversion loses qualifiers
```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 6 of 6).

```
// Fig. 7.2: fig07_02.cpp
// Using a member initializer to initialize a
// constant of a built-in data type.

#include <iostream.h>

class Increment {
public:
   Increment( int c = 0, int i = 1 );
   void addIncrement() { count += increment; }
   void print() const;

private:
   int count;
   const int increment;// const data member
};

// Constructor for class Increment
Increment::Increment( int c, int i )
   : increment( i )   // initializer for const member
{ count = c; }

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;
}

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 0; j < 3; j++ ) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

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

Fig. 7.2 Using a member initializer to initialize a constant of a built-in data type.

```
// Fig. 7.3: fig07_03.cpp
// Attempting to initialize a constant of
// a built-in data type with an assignment.
#include <iostream.h>

class Increment {
public:
   Increment( int c = 0, int i = 1 );
   void addIncrement() { count += increment; }
   void print() const;
private:
   int count;
   const int increment;
};

// Constructor for class Increment
Increment::Increment( int c, int i )
{              // Constant member 'increment' is not initialized
   count = c;
   increment = i;  // ERROR: Cannot modify a const object
}

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;
}

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 0; j < 3; j++ ) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

Fig. 7.3 Erroneous attempt to initialize a constant of a built-in data type by assignment (part 1 of 2).

```
Compiling...
Fig7_3.cpp
Fig7_3.cpp(18) : error: 'increment' :
   must be initialized in constructor base/member
   initializer list
Fig7_3.cpp(20) : error: l-value specifies const object
```

Fig. 7.3 Erroneous attempt to initialize a constant of a built-in data type by assignment (part 2 of 2).

```
// Fig. 7.4: date1.h
// Declaration of the Date class.
// Member functions defined in date1.cpp
#ifndef DATE1_H
#define DATE1_H

class Date {
public:
   Date( int = 1, int = 1, int = 1900 ); // default constructor
   void print() const;  // print date in month/day/year format
   ~Date();  // provided to confirm destruction order
private:
   int month;  // 1-12
   int day;    // 1-31 based on month
   int year;   // any year

   // utility function to test proper day for month and year
   int checkDay( int );
};

#endif
```

Fig. 7.4 Using member-object initializers (part 1 of 6).

```
// Fig. 7.4: date.cpp
// Member function definitions for Date class.
#include <iostream.h>
#include "date1.h"

// Constructor: Confirm proper value for month;
// call utility function checkDay to confirm proper
// value for day.
Date::Date( int mn, int dy, int yr )
{
   if ( mn > 0 && mn <= 12 )       // validate the month
      month = mn;
   else {
      month = 1;
      cout << "Month " << mn << " invalid. Set to month 1.\n";
   }

   year = yr;                      // should validate yr
   day = checkDay( dy );           // validate the day

   cout << "Date object constructor for date ";
   print();         // interesting: a print with no arguments
   cout << endl;
}

// Print Date object in form  month/day/year
```

```
void Date::print() const
   { cout << month << '/' << day << '/' << year; }

// Destructor: provided to confirm destruction order
Date::~Date()
{
   cout << "Date object destructor for date ";
   print();
   cout << endl;
}

```

Fig. 7.4 Using member-object initializers (part 2 of 6).

```
// Utility function to confirm proper day value
// based on month and year.
// Is the year 2000 a leap year?
int Date::checkDay( int testDay )
{
   static const int daysPerMonth[ 13 ] =
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
      return testDay;

   if ( month == 2 &&      // February: Check for leap year
        testDay == 29 &&
        ( year % 400 == 0 ||                      // year 2000?
         ( year % 4 == 0 && year % 100 != 0 ) ) ) // year 2000?
      return testDay;

   cout << "Day " << testDay << " invalid. Set to day 1.\n";

   return 1;  // leave object in consistent state if bad value
}
```

Fig. 7.4 Using member-object initializers (part 3 of 6).

```
// Fig. 7.4: emply1.h
// Declaration of the Employee class.
// Member functions defined in emply1.cpp
#ifndef EMPLY1_H
#define EMPLY1_H

#include "date1.h"

class Employee {
public:
   Employee( char *, char *, int, int, int, int, int, int );
   void print() const;
   ~Employee();  // provided to confirm destruction order
private:
   char firstName[ 25 ];
   char lastName[ 25 ];
   const Date birthDate;
   const Date hireDate;
};

#endif
```

Fig. 7.4 Using member-object initializers (part 4 of 6).

```
// Fig. 7.4: emply1.cpp
// Member function definitions for Employee class.
#include <iostream.h>
#include <string.h>
#include "emply1.h"
#include "date1.h"

Employee::Employee( char *fname, char *lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
   : birthDate( bmonth, bday, byear ),
     hireDate( hmonth, hday, hyear )
{
   // copy fname into firstName and be sure that it fits
   int length = strlen( fname );
   length = ( length < 25 ? length : 24 );
   strncpy( firstName, fname, length );
   firstName[ length ] = '\0';

   // copy lname into lastName and be sure that it fits
   length = strlen( lname );
   length = ( length < 25 ? length : 24 );
   strncpy( lastName, lname, length );
   lastName[ length ] = '\0';

   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
   cout << lastName << ", " << firstName << "\nHired: ";
   hireDate.print();
   cout << "  Birth date: ";
   birthDate.print();
   cout << endl;
}

// Destructor: provided to confirm destruction order
Employee::~Employee()
{
   cout << "Employee object destructor: "
        << lastName << ", " << firstName << endl;
}
```

Fig. 7.4 Using member-object initializers (part 5 of 6).

```
// Fig. 7.4: fig07_04.cpp
// Demonstrating composition: an object with member objects.
#include <iostream.h>
#include "emply1.h"

int main()
{
   Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );

   cout << '\n';
   e.print();

   cout << "\nTest Date constructor with invalid values:\n";
   Date d( 14, 35, 1994 );  // invalid Date values
   cout << endl;
   return 0;
}
```

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

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

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

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

Fig. 7.4 Using member-object initializers (part 6 of 6).

```
// Fig. 7.5: fig07_05.cpp
// Friends can access private members of a class.
#include <iostream.h>

// Modified Count class
class Count {
   friend void setX( Count &, int ); // friend declaration
public:
   Count() { x = 0; }                // constructor
   void print() const { cout << x << endl; }  // output
private:
   int x;  // data member
};

// Can modify private data of Count because
// setX is declared as a friend function of Count
void setX( Count &c, int val )
{
   c.x = val;  // legal: setX is a friend of Count
}

int main()
{
   Count counter;

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

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

Fig. 7.5 Friends can access `private` members of a class.

```
// Fig. 7.6: fig07_06.cpp
// Non-friend/non-member functions cannot access
// private data of a class.
#include <iostream.h>

// Modified Count class
class Count {
public:
  Count() { x = 0; }                   // constructor
  void print() const { cout << x << endl; }  // output
private:
   int x;  // data member
};

// Function tries to modify private data of Count,
// but cannot because it is not a friend of Count.
void cannotSetX( Count &c, int val )
{
   c.x = val;  // ERROR: 'Count::x' is not accessible
}

int main()
{
   Count counter;

   cannotSetX( counter, 3 ); // cannotSetX is not a friend
   return 0;
}
```

```
Compiling...
Fig07_06.cpp
Fig07_06.cpp(19) : error: 'x' :
   cannot access private member declared in class 'Count'
```

Fig. 7.6 Non-`friend`/non-member functions cannot access `private` class members.

```
// Fig. 7.7: fig07_07.cpp
// Using the this pointer to refer to object members.
#include <iostream.h>

class Test {
public:
   Test( int = 0 );             // default constructor
   void print() const;
private:
   int x;
};

Test::Test( int a ) { x = a; }  // constructor

```

Fig. 7.7 Using the `this` pointer (part 1 of 2).

```
void Test::print() const   // ( ) around *this required
{
   cout << "        x = " << x
        << "\n  this->x = " << this->x
        << "\n(*this).x = " << ( *this ).x << endl;
}

int main()
{
   Test testObject( 12 );

   testObject.print();

   return 0;
}
```

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

Fig. 7.7 Using the `this` pointer (part 2 of 2).

```
// Fig. 7.8: time6.h
// Cascading member function calls.

// Declaration of class Time.
// Member functions defined in time6.cpp
#ifndef TIME6_H
#define TIME6_H

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

   // set functions
   Time &setTime( int, int, int ); // set hour, minute, second
   Time &setHour( int );    // set hour
   Time &setMinute( int );  // set minute
   Time &setSecond( int );  // set second

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

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard() const;  // print standard time
private:
   int hour;              // 0 - 23
   int minute;            // 0 - 59
   int second;            // 0 - 59
};

#endif
```

Fig. 7.8 Cascading member function calls (part 1 of 4).

```
// Fig. 7.8: time.cpp
// Member function definitions for Time class.
#include "time6.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
Time &Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );
   return *this;   // enables cascading
}

// Set the hour value
Time &Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;

   return *this;   // enables cascading
}

// Set the minute value
Time &Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0;

   return *this;   // enables cascading
}

// Set the second value
Time &Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0;

   return *this;   // enables cascading
}

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

// Get the minute value
int Time::getMinute() const { return minute; }

```

Fig. 7.8 Cascading member function calls (part 2 of 4).

```
// Get the second value
int Time::getSecond() const { return second; }

// Display military format time: HH:MM
void Time::printMilitary() const
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard() const
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 7.8 Cascading member function calls (part 3 of 4).

```
// Fig. 7.8: fig07_08.cpp
// Cascading member function calls together
// with the this pointer
#include <iostream.h>
#include "time6.h"

int main()
{
   Time t;

   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
   cout << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";
   t.setTime( 20, 20, 20 ).printStandard();
   cout << endl;

   return 0;
}
```

```
Military time: 18:30
Standard time: 6:30:22 PM

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

Fig. 7.8 Cascading member function calls (part 4 of 4).

```
// Fig. 7.9: employ1.h
// An employee class
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
   Employee( const char*, const char* );  // constructor
   ~Employee();                          // destructor
   const char *getFirstName() const;     // return first name
   const char *getLastName() const;      // return last name

   // static member function
   static int getCount();  // return # objects instantiated

private:
   char *firstName;
   char *lastName;

   // static data member
   static int count;  // number of objects instantiated
};

#endif
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 1 of 5).

```
// Fig. 7.9: employ1.cpp
// Member function definitions for class Employee
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ1.h"

// Initialize the static data member
int Employee::count = 0;

// Define the static member function that
// returns the number of employee objects instantiated.
int Employee::getCount() { return count; }

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 );   // ensure memory allocated
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );    // ensure memory allocated
   strcpy( lastName, last );

   ++count;  // increment static count of employees
   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called." << endl;
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory
   --count;  // decrement static count of employees
}

// Return first name of employee
const char *Employee::getFirstName() const
{
   // const before return type prevents client modifying
   // private data. Client should copy returned string before
   // destructor deletes storage to prevent undefined pointer.
   return firstName;
}
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 2 of 5).

```

// Return last name of employee
const char *Employee::getLastName() const
{
   // const before return type prevents client modifying
   // private data. Client should copy returned string before
   // destructor deletes storage to prevent undefined pointer.
   return lastName;
}
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 3 of 5).

```
// Fig. 7.9: fig07_09.cpp
// Driver to test the Employee class
#include <iostream.h>
#include "employ1.h"

int main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name

   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();

   cout << "\n\nEmployee 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr;   // recapture memory
   e1Ptr = 0;
   delete e2Ptr;   // recapture memory
   e2Ptr = 0;

   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   return 0;
}
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 4 of 5).

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

Employee 1: Susan Baker
Employee 2: Robert Jones

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

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 5 of 5).

```
// Fig. 7.10: implementation.h
// Header file for class Implementation

class Implementation {
   public:
      Implementation( int v ) { value = v; }
      void setValue( int v ) { value = v; }
      int getValue() const { return value; }

   private:
      int value;
};
```

Fig. 7.10 Implementing a proxy class (part 1 of 4).

```
// Fig. 7.10: interface.h
// Header file for interface.cpp
class Implementation;   // forward class declaration

class Interface {
   public:
      Interface( int );
      void setValue( int );  // same public interface as
      int getValue() const;  // class Implementation
   private:
      Implementation *ptr;   // requires previous
                             // forward declaration
};
```

Fig. 7.10 Implementing a proxy class (part 2 of 4).

```
// Fig. 7.10: interface.cpp
// Definition of class Interface
#include "interface.h"
#include "implementation.h"

Interface::Interface( int v )
   : ptr ( new Implementation( v ) ) { }

// call Implementation's setValue function
void Interface::setValue( int v ) { ptr->setValue( v ); }

// call Implementation's getValue function
int Interface::getValue() const { return ptr->getValue(); }
```

Fig. 7.10 Implementing a proxy class (part 3 of 4).

```
// Fig. 7.10: fig07_10.cpp
// Hiding a class’s private data with a proxy class.
#include <iostream.h>
#include "interface.h"

int main()
{
   Interface i( 5 );

   cout << "Interface contains: " << i.getValue()
        << " before setValue" << endl;
   i.setValue( 10 );
   cout << "Interface contains: " << i.getValue()
        << " after setValue" << endl;
   return 0;
}
```

```
Interface contains: 5 before setVal
Interface contains: 10 after setVal
```

Fig. 7.10 Implementing a proxy class (part 4 of 4).