C++ Basics
Classes

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

```cpp
#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** <u>**local objects**</u> 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**. <u>Destructors are **NOT** called</u>.

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
```

```cpp
#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 can improve performance— 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 *changes a data member* of the class on the same object is a compilation error.

- Invoking a non-const member function on a const object is a compilation error.

```cpp
3  #include "Time.h" // include Time class definition
4
5  int main() {
6     Time wakeUp{6, 45, 0}; // non-constant object
7     const Time noon{12, 0, 0}; // constant object
8
9                                // OBJECT      MEMBER FUNCTION
10    wakeUp.setHour(18);        // non-const   non-const
11    noon.setHour(12);          // const       non-const
12    wakeUp.getHour();          // non-const   const
13    noon.getMinute();          // const       const
14    noon.toUniversalString();  // const       const
15    noon.toStandardString();   // const       non-const
16 }
```

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 <u>the constructor *completes* initialization of the object until that object's destructor is called</u>.

Composition

An **AlarmClock** object needs to know when it's supposed to sound its alarm, so why not include a **Time** object as a <u>member</u> of the **AlarmClock** class?

Such a software-reuse capability is called **composition** (or **aggregation**) and is sometimes referred to as a ***has-a*** **relationship -** <u>*a class can have objects of other classes as members*</u>*.*

```
class AlarmClock {
…
private:
   Time _timeToAlarm;
};
```

Class *AlarmClock* <u>**has-a**</u> *Time* object**.**

Composition

- Data members are constructed in the order in which they're **declared in the class definition** (NOT in the order they're listed in the constructor's member-initializer list) and **before** their enclosing class objects are constructed.

```
class B {
public:
   B(int val=0)
      : _val(val)
   {
      std::cout << "B::B(" << _val << ") called" << std::endl;
   }
   ~B() {
      std::cout << "B::~B(" << _val << ") called" << std::endl;
   }
private:
   int _val;
};
```

The **colon (:)** following the constructor's header begins the member-initializer list.

Composition

```
class A {
public:
   A()
      : _b2(2),
        _b1(1)
   {
      std::cout << "A::A called" << std::endl;
   }
   ~A() {
      std::cout << "A::~A called" << std::endl;
   }
private:
   B _b1;
   B _b2;
};
```

Composition

```
int main()
{
   A a;
   return 0;
}
```

Result:
B::B(1) called
B::B(2) called
A::A called
A::~A called
B::~B(2) called
B::~B(1) called

- For clarity, list the member initializers in the order that the class's data members are declared.

Composition

Result:
B::B(1) called
B::B(2) called
A::A called
A::~A called
B::~B(2) called
B::~B(1) called

This confirms that **objects are constructed from the *inside out*** and **destructed in the *reverse* order**, from the ***outside in.***

The constructor builds a class object ''from the bottom up'':
[1] it invokes the member constructors, and
[2] finally, it executes its own body.

A destructor destructs an object in the reverse order:
[1] first, the destructor executes its own body,
[2] then, it invokes its member destructors

Composition (Example)

```
#include <string>

#ifndef DATE_H
#define DATE_H

class Date {
public:
   static const unsigned int monthsPerYear{12}; // months in a year
   explicit Date(unsigned int = 1, unsigned int = 1, unsigned int = 1900);
   std::string toString() const; // date string in month/day/year format
   ~Date(); // provided to confirm destruction order
private:
   unsigned int month; // 1-12 (January-December)
   unsigned int day; // 1-31 based on month
   unsigned int year; // any year

   // utility function to check if day is proper for month and year
   unsigned int checkDay(int) const;
};

#endif
```

```
#include <array>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include "Date.h" // include Date class definition
using namespace std;

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
Date::Date(unsigned int mn, unsigned int dy, unsigned int yr)
   : month{mn}, day{checkDay(dy)}, year{yr} {
   if (mn < 1 || mn > monthsPerYear) { // validate the month
      throw invalid_argument("month must be 1-12");
   }

   // output Date object to show when its constructor is called
   cout << "Date object constructor for date " << toString() << endl;
}

// print Date object in form month/day/year
string Date::toString() const {
   ostringstream output;
   output << month << '/' << day << '/' << year;
   return output.str();
}

// output Date object to show when its destructor is called
Date::~Date() {
   cout << "Date object destructor for date " << toString() << endl;
}
```

```
// utility function to confirm proper day value based on
// month and year; handles leap years, too
unsigned int Date::checkDay(int testDay) const {
   static const array<int, monthsPerYear + 1> daysPerMonth{
      0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   // determine whether testDay is valid for specified month
   if (testDay > 0 && testDay <= daysPerMonth[month]) {
      return testDay;
   }

   // February 29 check for leap year
   if (month == 2 && testDay == 29 && (year % 400 == 0 ||
      (year % 4 == 0 && year % 100 != 0))) {
      return testDay;
   }

   throw invalid_argument("Invalid day for current month and year");
}
```

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition

class Employee {
public:
   Employee(const std::string&, const std::string&,
      const Date&, const Date&);
   std::string toString() const;
   ~Employee(); // provided to confirm destruction order
private:
   std::string firstName; // composition: member object
   std::string lastName; // composition: member object
   const Date birthDate; // composition: member object
   const Date hireDate; // composition: member object
};

#endif
```

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

// constructor uses member initializer list to pass initializer
// values to constructors of member objects
Employee::Employee(const string& first, const string& last,
   const Date &dateOfBirth, const Date &dateOfHire)
   : firstName{first}, // initialize firstName
     lastName{last}, // initialize lastName
     birthDate{dateOfBirth}, // initialize birthDate
     hireDate{dateOfHire} { // initialize hireDate
   // output Employee object to show when constructor is called
   cout << "Employee object constructor: "
      << firstName << ' ' << lastName << endl;
}

// print Employee object
string Employee::toString() const {
   ostringstream output;
   output << lastName << ", " << firstName << "  Hired: "
      << hireDate.toString() << "  Birthday: " << birthDate.toString();
   return output.str();
}
```

Notice, that **Employee** constructor uses the <u>compiler-provided *default copy constructor*</u> for **Date** that copies each data member of the constructor's argument object into the corresponding member of the object being initialized.

```cpp

// output Employee object to show when its destructor is called
Employee::~Employee() {
   cout << "Employee object destructor: "
      << lastName << ", " << firstName << endl;
}
```

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

int main() {
   Date birth{7, 24, 1949};
   Date hire{3, 12, 1988};
   Employee manager{"Bob", "Blue", birth, hire};

   cout << "\n" << manager.toString() << endl;
}
```

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

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

Composition

What Happens When You Do Not Use the Member-Initializer List?

- If a member object is *not* initialized through a member initializer, the member object's *default constructor* will be called *implicitly*.
- Values, if any, established by the default constructor can be overridden by ***set*** functions.

❖ Initialize member objects explicitly through member initializers. This eliminates the overhead of "**doubly initializing**" member objects—*once* when the member object's default constructor is called and *again* when set functions are called in the constructor body (or later) to initialize the member object.

Composition

```
class B {
public:
   B(int val=0)
      : _val(val)
   {
      std::cout << "B::B(" << _val << ") called" << std::endl;
   }
   ~B() {
      std::cout << "B::~B(" << _val << ") called" << std::endl;
   }
   void setVal(int val) {
      std::cout << "B::setVal(" << val << ") called" << std::endl;
      _val = val;
   }
private:
   int _val;
};
```

Composition

```
class A {
public:
   A()
//       : _b2(2),
//         _b1(1)
   {
      std::cout << "A::A called" << std::endl;
      // these two functions cause 'double initialization'
      _b1.setVal(1);
      _b2.setVal(2);
   }
   ~A() {
      std::cout << "A::~A called" << std::endl;
   }
private:
   B _b1;
   B _b2;
};
```

Composition

```
int main()
{
   A a;
    // A should have these two functions if member-initializer list is skipped
   //a.setB1Val(1);
   //a.setB2Val(2);
   return 0;
}
```

Result:

B::B(0) called
B::B(0) called
A::A called
B::setVal(1) called
B::setVal(2) called
A::~A called
B::~B(2) called
B::~B(1) called

Composition

What Happens When You Do Not Use the Member-Initializer List?

- A *compilation error* occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

- If a data member is an object of another class, making that member object public does not violate the encapsulation and hiding of that member object's private members. But, it does violate the encapsulation and hiding of the enclosing class's implementation, so member objects of class types **should still be private**.

friend Functions and friend Classes

A **friend function** of a class is a non-member function that **has the right to access** the public and non-public class members.

Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.

To declare a function as a friend of a class, place the function prototype in the class definition and precede it with the keyword **friend**:

```
class classOne {
   friend class ClassTwo;
   friend void ClassThree::someFunction(int);
   friend void someGlobalFunction(classOne&);
   friend void anotherGlobalFunction();
…
};
```

- All member functions of ClassTwo will become friends of ClassOne.
- Only the someFunction from ClassThree will become friend of ClassOne.
- The someGlobalFunction and anotherGlobalFunction will be able **to access the non-public members** of classOne.

friend Functions

```
3  #include <iostream>
4  using namespace std;
5
6  // Count class definition
7  class Count {
8     friend void setX(Count&, int); // friend declaration
9  public:
10    int getX() const {return x;}
11 private:
12    int x{0};
13 };
14
15 // function setX can modify private data of Count
16 // because setX is declared as a friend of Count (line 8)
17 void setX(Count& c, int val) {
18    c.x = val; // allowed because setX is a friend of Count
19 }
20
21 int main() {
22    Count counter; // create Count object
23
24    cout << "counter.x after instantiation: " << counter.getX() << endl;
25    setX(counter, 8); // set x using a friend function
26    cout << "counter.x after call to setX friend function: "
27       << counter.getX() << endl;
28 }
```

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

<u>friend</u> member Functions

<u>a.h</u>

```
#ifndef A_H
#define A_H

class Count;
class A {
public:
   void changeCount(Count& c);
};

#endif // A_H
```

<u>a.cpp</u>

```
#include "a.h"
#include "count.h"

void A::changeCount(Count& c) {
   c.x = 12;
}
```

friend member Functions

count.h

```
#ifndef COUNT_H
#define COUNT_H

#include "a.h"

// Count class definition
class Count {
  friend void setX(Count&, int); // friend declaration
  friend void A::changeCount(Count&);
public:
  int getX() const { return x; }
private:
  int x{0};
};

#endif // COUNT_H
```

friend member Functions

```
#include "count.h"

void setX(Count& c, int val) {
  c.x = val; // allowed because setX is a friend of Count
}

int main() {
  Count counter; // create Count object

  cout << "counter.x after instantiation: " << counter.getX() << endl;
  setX(counter, 8); // set x using a friend function
  cout << "counter.x after call to setX friend function: "
    << counter.getX() << endl;

  A a;
  a.changeCount(counter);
  cout << "counter.x after changeCount: " << counter.getX() << endl;
}
```

Result: **counter.x after instantiation: 0**
counter.x after call to setX friend function: 8
counter.x after changeCount: 12

friend Functions and friend Classes

- The friend declaration(s) can appear *anywhere* in a class and **are not affected by access specifiers** public, protected or private.

- For class B to be a friend of class A, class A must ***explicitly*** **declare** that class B is its friend.

- Friendship **is not *symmetric***—if class A is a friend of class B, you cannot infer that class B is a friend of class A.

- Friendship **is not transitive**—if class A is a friend of class B and class B is a friend of class C, you cannot infer that class A is a friend of class C.

- Even though the prototypes for friend functions appear in the class definition, **friends are not member functions**.

❖ It is a good programming practice to place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

this pointer

There's only one copy of each class's functionality, but there can be many objects of a class, so how do member functions know which object's data members to manipulate?

- Every object has access to its own address through a pointer called **this** (a C++ keyword).
- The 'this' pointer is ***NOT* part of the object itself** - the memory occupied by the 'this' pointer is NOT reflected in the result of a **sizeof** operation on the object.
- The 'this' pointer is passed (by the compiler) as an ***implicit* argument to each of the object's non-static member functions**.

<u>this</u> pointer

Member functions use the 'this' pointer ***implicitly*** (as we've done so far) or ***explicitly*** to reference an object's data members and other member functions.

A common *explicit* use of the 'this' pointer is to avoid ***naming conflicts*** between a <u>class's data members</u> and <u>member-function parameters (or other local variables)</u>:

```
// set hour value
void Time::setHour(int hour) {
   if (hour >= 0 && hour < 24) {
      this->hour = hour; // use this-> to access data member
   }
   else {
      throw invalid_argument("hour must be 0-23");
   }
}
```

Here, the local variable is said to ***hide*** or ***shadow*** the data member.

this pointer

The **type** of the ‘this’ pointer depends on the type of the object and whether the member function in which ‘this’ is used is declared const:

- In a non-const member function of class Employee, the ‘this’ pointer has the type **Employee* const**—a constant pointer to a *nonconstant* Employee.

- In a const member function, ‘this’ has the type **const Employee* const**—a constant pointer to a *constant* Employee.

<u>this</u> pointer

```
class Test {
public:
   explicit Test(int);
   void print() const;
private:
   int x{0};
};

// constructor
Test::Test(int value) : x{value} {} // initialize x to value

// print x using implicit then explicit this pointers;
// the parentheses around *this are required
void Test::print() const {
   // implicitly use the this pointer to access the member x
   cout << "        x = " << x;

   // explicitly use the this pointer and the arrow operator
   // to access the member x
   cout << "\n  this->x = " << this->x;

   // explicitly use the dereferenced this pointer and
   // the dot operator to access the member x
   cout << "\n(*this).x = " << (*this).x << endl;
}
```

this pointer

```
int main() {
   Test testObject{12}; // instantiate and initialize testObject
   testObject.print();
}
```

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

Note the parentheses around *this when used with the dot member-selection operator (.).

The parentheses are required because the dot(.) operator has **higher precedence** than the * operator.

Without the parentheses, the expression ***this.x** would be evaluated as if it were parenthesized as ***(this.x)**, which is a *compilation error*.

<u>this</u> pointer

Another use of the ‘this’ pointer is to enable **cascaded member-function calls**—that is, invoking multiple functions sequentially in the same statement:

```
Time t;
t.setHour(10).setMinute(20).setSecond(30);
```

this pointer

```cpp
#include <string>

// Time class definition.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

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

   // set functions (the Time& return types enable cascading)
   Time& setTime(int, int, int); // set hour, minute, second
   Time& setHour(int); // set hour
   Time& setMinute(int); // set minute
   Time& setSecond(int); // set second

   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
```

this pointer

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

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

// set hour value
Time& Time::setHour(int h) { // note Time& return
```

```
    if (h >= 0 && h < 24) {
        hour = h;
    }
    else {
        throw invalid_argument("hour must be 0-23");
    }

    return *this; // enables cascading
}

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

    return *this; // enables cascading
}

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

    return *this; // enables cascading
}
```

this pointer

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

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

   t.setHour(18).setMinute(30).setSecond(22); // cascaded function calls

   // output time in universal and standard formats
   cout << "Universal time: " << t.toUniversalString()
      << "\nStandard time: " << t.toStandardString();

   // cascaded function calls
   cout << "\n\nNew standard time: "
      << t.setTime(20, 20, 20).toStandardString() << endl;
}
```

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

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

this pointer

The dot operator (.) associates from left to right

```
t.setHour(10).setMinute(20).setSecond(30);
```

`t.setHour(10)` returns the **updated** object t as the value of this function call.
The remaining expression is then interpreted as

```
t.setMinute(20).setSecond(30);
```

then

```
t.setSecond(30);
```

static class members

There is an **important exception to the rule** that each object of a class has its own copy of all the data members of the class.

In certain cases, only *one* copy of a variable should be *shared* by *all* objects of a class.

A **static data member** is used for these and other reasons.

Such a variable represents "classwide" information, i.e., data that is shared by *all* instances and **is *not* specific to any one object of the class**.

- ❖ Use static data members **to save storage** when a single copy of the data for all objects of a class will suffice—such as a constant that can be shared by all objects of the class.

static class members (example)

```
class Employee {
public:
   Employee(const std::string&, const std::string&); // constructor
   ~Employee(); // destructor
   std::string getFirstName() const; // return first name
   std::string getLastName() const; // return last name

   // static member function
   static unsigned int getCount(); // return # of objects instantiated
private:
   std::string firstName;
   std::string lastName;

   // static data
   static unsigned int count; // number of objects instantiated
};
```

```cpp
3 #include <iostream>
4 #include "Employee.h" // Employee class definition
5 using namespace std;
6
7 // define and initialize static data member at global namespace scope
8 unsigned int Employee::count{0}; // cannot include keyword static
9
10 // define static member function that returns number of
11 // Employee objects instantiated (declared static in Employee.h)
12 unsigned int Employee::getCount() {return count;}
13
14 // constructor initializes non-static data members and
15 // increments static data member count
16 Employee::Employee(const string& first, const string& last)
17    : firstName(first), lastName(last) {
18    ++count; // increment static count of employees
19    cout << "Employee constructor for " << firstName
20       << ' ' << lastName << " called." << endl;
21 }
22
23 // destructor decrements the count
24 Employee::~Employee() {
25    cout << "~Employee() called for " << firstName
26       << ' ' << lastName << endl;
27    --count; // decrement static count of employees
28 }
29
30 // return first name of employee
31 string Employee::getFirstName() const {return firstName;}
32
33 // return last name of employee
34 string Employee::getLastName() const {return lastName;}
```

Note, that the static keyword **must not be used** in line 8 and 12.

static class members (example)

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

int main() {
   // no objects exist; use class name and binary scope resolution
   // operator to access static member function getCount
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   // the following scope creates and destroys
   // Employee objects before main terminates
   {
      Employee e1{"Susan", "Baker"};
      Employee e2{"Robert", "Jones"};

      // two objects exist; call static member function getCount again
      // using the class name and the scope resolution operator
      cout << "Number of employees after objects are instantiated is "
         << Employee::getCount();

      cout << "\n\nEmployee 1: "
         << e1.getFirstName() << " " << e1.getLastName()
         << "\nEmployee 2: "
         << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
   }
```

Note, that instead of Employee::getCount(), we can write **e1.getCount()** or **e2.getCount()**

static class members (example)

```
    // no objects exist, so call static member function getCount again
    // using the class name and the scope resolution operator
    cout << "\nNumber of employees after objects are deleted is "
       << Employee::getCount() << endl;
}
```

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

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```

static class members

A class's static data members have ***class scope.***

A static data member ***must*** **be initialized *exactly* once**.

A class's static data members and static member functions **exist and can be used even if no objects of that class have been instantiated**.

To access a private or protected static class member when *no* objects of the class exist, provide a public **static member function** and call the function by prefixing its name with the class name and scope resolution operator, f.e.

Employee::getCount();

A static member function **is a service of the *class*, *not* of a specific *object*** of the class.

It is good programming practice to initialize the class static data members ***at global namespace scope:***

```
unsigned int Employee::count{0}; // in Employee.cpp
```