UEE1303
Objective-Oriented Programming

C++_Lecture 04:

Classes and Objects: Advanced Topics (I)

C: How to Program 8th ed.

Agenda

- When Constructors and Destructors are Called (chapter 17.7)
- Passing to and Returning Objects from Functions Copy Constructor
- Utility Function (chapter 17.4)
- Default Memberwise Assignment (chapter 17.9)

When Constructors and Destructors are Called

- Constructors and destructors are called implicitly by the compiler.
- 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.

Example

```
// Fig. 17.7: CreateAndDestroy.h
// CreateAndDestroy class definition.
// Member functions defined in CreateAndDestroy.cpp.
#include <string>
using namespace std;
#ifndef CREATE_H
#define CREATE_H
class CreateAndDestroy
{
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy
#endif
```

Example (cont.)

```
// Fig. 17.8: CreateAndDestroy.cpp
// CreateAndDestroy class member-function definitions.
#include <iostream>
#include "CreateAndDestroy.h"
using namespace std;

// constructor
CreateAndDestroy::CreateAndDestroy( int ID, string
messageString )
{
   objectID = ID; // set object's ID number
   message = messageString; // set object's descriptive
message

   cout << "Object " << objectID << "   constructor runs "
        << message << endl;
} // end CreateAndDestroy constructor
```

Example (cont.)

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

    cout << "Object " << objectID << "   destructor runs "
         << message << endl;
} // end ~CreateAndDestroy destructor
```

Example (cont.)

```
// Fig. 17.9: fig17_9.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
#include "CreateAndDestroy.h"
using namespace std;

void create( void ); // prototype
// global object
CreateAndDestroy first( 1, "(global before main)" );
int main()
{
    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
    CreateAndDestroy second( 2, "(local automatic in
main)" );
    static CreateAndDestroy third( 3, "(local static in
main)" );
   create(); // call function to create objects
```

Example (cont.)

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

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

Example (cont.)

- screen output

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

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

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

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

Example (cont.)

- screen output

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

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

Passing Objects to Functions

- When passing an object to a function, its copy is created in memory.
  - It does not, however, invoke a class constructor.
    - The primary purpose of a constructor function is to initialize a newly created object.
- When a function that has objects as parameters terminates, the copies of the objects that are passed to the function are destroyed at that point.
  - The class destructor is called as many times as there are copies to destroy.

Example of Passing Objects to Functions

```
#include <iostream>
using namespace std;
int bcount = 0;
class Box {
public:
    Box( float s1, float s2, float s3) { // constructor
        side1 = s1, side2 = s2, side3 = s3;
        bcount++;
    }
    ~Box() {
        bcount--;
        cout << "\t\tBox destroyed!\n";
    }
    float getVolumn() {
        return side1 * side2 * side3;
    }
private:
    float side1, side2, side3;
}; // end class Box
```

Example of Passing Objects to Functions (cont.)

```
float addBoxes(Box, Box);
int main()
{
    Box a(1, 2, 3), b(2, 3, 4);
    cout << bcount << "boxes built so far.\n";
    cout << "\t\tTotal volumn = " << addBoxes(a, b) <<
endl;
    cout << bcount << "boxes built so far.\n";
    return 0;
}
float addBoxes(Box b1, Box b2)
{
    cout << "\t\tAdding boxes:\n";
    cout << "\t\t" << bcount << "boxes built so far.\n";

    return b1.getVolumn()+b2.getVolumn();
}
```

Example of Passing Objects to Functions (cont.)

- screen output

```
2 boxes built so far.
             Adding boxes:
             2 boxes built so far.
             Box destroyed!
             Box destroyed!
             Total volume = 30
0 boxes built so far.
             Box destroyed!
             Box destroyed!
```

Passing Objects to Functions (cont.)

- One way to prevent destructor calls when passing objects to functions is passing reference or addresses of the objects.
- Copies of the objects passed to the function will not be created in this case, and the destructor will not be called when the function terminates.

```
float addBoxes(Box &b1, Box &b2)
{
    cout << "\t\tAdding boxes:\n";
    cout << "\t\t" << bcount << "boxes built so far.\n";

    return b1.getVolumn()+b2.getVolumn();
}
```

Example2 of Passing Objects to Functions

```
#include <iostream>
using namespace std;
int bcount = 0;
class Box {
public:
    Box( float s1, float s2, float s3) { // constructor
        side1 = s1, side2 = s2, side3 = s3;
        bcount++;
    }
    ~Box() {
        bcount--;
        cout << "\t\tBox destroyed!\n";
    }
    float getVolumn() {
        return side1 * side2 * side3;
    }
private:
    float side1, side2, side3;
}; // end class Box
```

Example2 of Passing Objects to Functions (cont.)

```
float addBoxes(Box, Box);
int main()
{
    Box a(1, 2, 3), b(2, 3, 4);
    cout << bcount << "boxed built so far.\n";
    cout << "\t\tTotal volumn = " << addBoxes(a, b) <<
endl;
    cout << bcount << "boxed built so far.\n";
    return 0;
}
float addBoxes(Box &b1, Box &b2)
{
    cout << "\t\tAdding boxes:\n";
    cout << "\t\t" << bcount << "boxes built so far.\n";

    return b1.getVolumn()+b2.getVolumn();
}
```

Example2 of Passing Objects to Functions (cont.)

- screen output

```
2 boxes built so far.
             Adding boxes:
             2 boxes built so far.
             Total volume = 30
2 boxes built so far.
             Box destroyed!
             Box destroyed!
```

Return Objects from Functions

- A function may return an object by using the return statement.
  - The process of returning objects from functions is similar to the process of returning any other non-object value.

Example of Return Objects from Functions

```
#include <iostream>
using namespace std;
class Pixel
{
public:
    Pixel() {x = 0; y = 0;} // constructor
    ~Pixel() { cout << "\t\tPixel destroyed!\n";}
    void setXY(int x1, int y1) { x = x1; y = y1;}
    void getCoord() {
        cout << "Pixel's coordinates: \n";
        cout << "X = " << x << "Y = " << y << endl;
    }
    Pixel move_10(Pixel t) {
        t.x = t.x + 10;  t.y = t.y + 10;
        return t;
    }
private:
    int x, y;
}; // end class Pixel
```

Example of Return Objects from Functions (cont.)

```
Pixel setCoord() {
    int x1, y1;
    cout << "Enter x and y coordinates =>";
    cin >> x1 >> y1;
    Pixel temp;
    temp.setXY(x1, y1);
    return temp;
}
int main()
{
    Pixel p1, p2;
    p1 = setCoord();
    p1.getCoord();
    p2 = p1.move_10(p1);
    p2.getCoord();
    p1.getCoord();
    return 0;
}
```

Example of Return Objects from Functions (cont.)

- screen output

```
Enter x and y coordinates => 10, -15
              Pixel destroyed!
              Pixel destroyed!
Pixel's coordinates:
X = 10, Y = -15
              Pixel destroyed!
              Pixel destroyed!
Pixel's coordinates:
X = 20, Y = -5
Pixel's coordinates:
X = 10, Y = -15
              Pixel destroyed!
              Pixel destroyed!
```

setCoord():
Destroy temp
Destroy returned object's copy

move_10():
Destroy t
Destroy returned object's copy

main():
Destroy p2
Destroy p1

Copy Constructor

- Regular constructor is called when a new object is instantiated.
- Copy constructor is called when an object is copied.
  - An object is used to initialize another object in a declaration statement

    ex:
```
Pixel p3(p2);   // call copy constructor
Pixel p2 = p1; // call copy constructor
(x) p2 = p4;   // Default Memberwise Assignment
```
  - An object is passed to a function

    ex: `fun1(p1);`
  - An object is returned from a function

    ex: `p2 = fun2();`

Copy Constructor

- General Format

```
⟨CNAME⟩:: ⟨CNAME⟩(const ⟨CNAME⟩ & ⟨var⟩)
{
    //copy constructor: function body
};
```

  - ⟨CNAME⟩ is the class name
  - ⟨var⟩ is the name for formal parameter of copied object
  - If no copy constructor is specified, the compiler automatically generate one

Example1 of Copy Constructors

```
class CStr
{
public:
    CStr(char* word); //A
    CStr(const CStr & old); //B
    void ShowCStr();
private:
    char * line;//default access is private
};
```

```
int main() {
    CStr one("test!"); //call A
    CStr two(one); //call B; CStr two = one;
    two.ShowCStr();
    return 0;
}
```

Example1 of Copy Constructors (cont.)

```
CStr::CStr(char * word) //A
{
    line = new char [strlen(word)+1];
    strcpy(line, word);
}

CStr::CStr(const CStr & old) //B
{
    line = new char [strlen(old.line)+1];
    strcpy(line, old.line);
}

void CStr::ShowCStr(){
    cout << line << endl;
}
```

Example2 of Copy Constructor

```
#include <iostream>
using namespace std;
class Pixel
{
public:
    Pixel(int a, int b) {// regular constructor
        x = a; y = b;
        cout << "\tNormal Constructor" << endl;
    }
    Pixel(const Pixel & p) {// copy constructor
        x = p.x; y = p.y;
        cout << "\tCopy Constructor" << endl;
    }
    ~Pixel() { cout << "\tDestructor" << endl;}
    void setX(int x1) { x = x1}
    void setY(int y1) { y = y1}
    void showXY () {
        cout << "X = " << x << "Y = " << y << endl;
    }
```

Example2 of Copy Constructor (cont.)

```
private:
    int x, y;
} // end class Pixel
Pixel center(Pixel tp) {
    tp.setX(512);
    tp.setY(512);
    return tp;
}
int main() {
    Pixel p1(10, 20);
    p1.showXY();
    Pixel p2 = p1;
    p2.showXY();
    p2 = center(p1);
    p2.showXY();
    return 0;
}
```

Example2 of Copy Constructor (cont.)

- screen output

```
       Normal Constructor
X = 10 Y = 20
       Copy Constructor
X = 10 Y = 20
       Copy Constructor
       Copy Constructor
       Destructor
       Destructor
X = 512 Y = 512
       Destructor
       Destructor
```

Default Copy Constructor

- If a class does not have an explicit copy constructor definition, the C++ compiler will create the default copy constructor.
  - an identical (bit-by-bit) copy of an object

- Assume an object contains a pointer that is used to dynamically allocate memory for that object.
  - If a bit-by-bit copy of the object is created, it will contain the pointer pointing the same memory location
  - When the object's copy is destroyed, the destructor will free the memory that is still being used by the original object.

Example3 of Copy Constructor

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class Message {
   char *mes;
public:
   Message(char *m);                  //regular constructor
   Message(const Message &cm);          //copy constructor
   ~Message();                          //destructor
   void printMes(){ cout<<"\tMessage: "<<mes<<endl; }
   char * getMes(){ return mes; }
};
```

Example3 of Copy Constructor (cont.)

```
Message::Message(char *m)      //regular constructor
{
   cout << "Normal Constructor Called!" << "  ";
   mes = new char[strlen(m)+1];
   cout << "mes ==> " << &mes << endl;
   if(!mes) {
      cout << "Allocation Error!";
      exit(1);
   }
   strcpy(mes, m);
}
```

Example3 of Copy Constructor (cont.)

```
Message::Message(const Message &cm) //copy constructor
{
   cout << "Copy Constructor Called!" << "    ";
   mes = new char[strlen(cm.mes)+1];
   cout << "mes ==> " << &mes << endl;
   if(!mes) {
      cout << "Allocation Error!";
      exit(1);
   }
   strcpy(mes, cm.mes);
}
Message::~Message()           //destructor
{
   cout << "Destructor Called!" << "          ";
   cout << "mes ==> " << &mes << "  " <<endl;
   delete []mes;
}
```

Example3 of Copy Constructor (cont.)

```
void upperMes(Message m)      //Modifies a message
{
   int n = strlen(m.getMes());
   char * t = m.getMes();
   for (int i = 0; i < n; i++)
      t[i] = toupper(t[i]);
   t[n] = '\0';
   Message temp(t);
   temp.printMes();
}
```

Example3 of Copy Constructor (cont.)

```
int main()
{
   Message m1("PROG5_5");
   m1.printMes( );
   Message m2("Demonstrates copy constructor!");
   Message m3 = m2;          //Calls the copy constructor
   m3.printMes();
   upperMes(m3);             //Calls the copy constructor
   return 0;
}
```

Example3 of Copy Constructor (cont.)

- screen output

```
Normal Constructor Called!  mes ==> 0x7fff9acd3920
        Message: PROG5_5
Normal Constructor Called!  mes ==> 0x7fff9acd3910
Copy Constructor Called!    mes ==> 0x7fff9acd3900
        Message: Demonstrates copy constructor!
Copy Constructor Called!    mes ==> 0x7fff9acd3930
Normal Constructor Called!  mes ==> 0x7fff9acd38b0
        Message: DEMONSTRATES COPY CONSTRUCTOR!
Destructor Called!          mes ==> 0x7fff9acd38b0
Destructor Called!          mes ==> 0x7fff9acd3930
Destructor Called!          mes ==> 0x7fff9acd3900
Destructor Called!          mes ==> 0x7fff9acd3910
Destructor Called!          mes ==> 0x7fff9acd3920
```

Utility Function

- A utility function (i.e., helper function) is not part of a class's public interface; rather, it's a private member function that supports the operation of the class's public member functions.
- Utility functions are not intended to be used by clients of a class (but can be used by friends of a class)

SalesPerson.h

```
// Fig. 17.5: SalesPerson.h (from 7th edition)
// SalesPerson class definition
// Member functions are defined in SalesPerson.cpp
#ifndef SALESP_H
#define SALESP_H
class SalesPerson
{
public:
    static const int monthsPerYear = 12;
    SalesPerson(); // constructor
    void getSalesFromUser(); // input sales from keyboard
    void setSales(int, double); // set sales for a
specific month
    void printAnnualSales(); // summarize and print sales
private:
    double totalAnnualSales();
    double sales[monthsPerYear];
}; // end class SalesPerson
#endif
```

SalesPerson.cpp

```
// Fig. 17.6: SalesPerson.cpp (from 7th edition)
#include <iostream>
#include <iomanip>
#include "SalesPerson.h"
using namespace std;
SalesPerson::SalesPerson() {// constructor
    for (int = 0; i < monthsPerYear; i++)
        sales[i] = 0;
}
// get 12 sales figures from the user
void SalesPerson::getSalesFromUser() {
    double salesFigure;
    for (int i = 1; i <= monthsPerYear; i++){
        cout << "Enter sales amount for month " << i
             << ":";
        cin >> salesFigure;
        setSales( i, salesFigure);
    }
}
```

SalesPerson.cpp (cont.)

```
// set sales for a specific month
void SalesPerson::setSales(int month, double amount) {
    if (month >= 1 && month <= monthsPerYear && amount > 0)
        sales[month-1] = amount;
    else
        cout << "Invalid month or sales figure" << endl;
}
// summarize and print sales with helper function
void SalesPerson::printAnnualSales() {
    cout << setprecision(2) << fixed
         << "\nThe total annual sales are:$"
         << totalAnnualSales() << endl;
}
double SalesPerson::totalAnnualSales() {
    double total = 0.0;
    for (int i = 0; i < monthsPerYear; i++)
        total += sales[i];
    return total;
}
```

Example of Utility Function

```
// Fig. 17.7: fig17_07.cpp (from 7th edition)
// Utility function demonstration.
#include "SalesPerson.h"

int main()
{
    SalesPerson s;  // create SalesPerson object s
    s.getSalesFromUser();
    s.printAnnualSales();

    return 0;
}
```

Example of Utility Function (cont.)

- screen output

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5839.22
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $6012.59
```

Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type.
- By default, such assignment is performed by memberwise 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.
- Caution: *Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory*

Date.h

```
// Fig. 17.13: Date.h
// Date class definition
// Member functions are defined in Date.cpp
#ifndef DATE_H
#define DATE_H
class Date
{
public:
    Date( int = 1, int = 1, int = 2000); // constructor
    void print();
private:
    int month;
    int day;
    int year;
}; // end class Date
#endif
```

Date.cpp

```
// Fig. 17.14: Date.cpp
#include <iostream>
#include "Date.h"
using namespace std;

Date::Date( int m, int d, int y)
{
    month = m;
    day = d;
    year = y;
}
void Date::print()
{
    cout << month << '/' << day << '/' << year;
}
```

fig17_19.cpp

```
// Fig. 17.15: fig17_15.cpp
// default memberwise assignment.
#include <iostream>
#include "date.h"
using namespace std;
int main()
{
    Date date1( 7, 4, 2004);
    Date date2;  // date2 defaults to 1/1/2000
    cout << "date1 = ";
    date1.print();
    cout << "\ndate2 = ";
    date2.print();
    date2 = date1; // default memberwise assignment
    cout << "\n\nAfter default memberwise assignment,
date2 = ";
    date2.print();
    return 0;
}
```

Summary

- Copy Constructor
- When Constructors and Destructors are Called Passing to and Returning Objects from Functions Copy Constructor
  - Constructors and destructors are called implicitly by the compiler.
  - Generally, destructor calls are made in the reverse order of the corresponding constructor calls.
- Utility Function
  - Supports the operation of the class's public member function
- Default Memberwise Assignment
  - The assignment operator (=) can be used to assign an object to another object of the same type.

References

- Paul Deitel and Harvey Deitel, "C How to Program" Eighth Edition
  - Chapter 17
- Paul Deitel and Harvey Deitel, "C++ How to Program (late objects version)" Seventh Edition
  - Chapter 9: Class
  - Chapter 10: Classes: A Deeper Look
- W. Savitch, "Absolute C++," Fourth Edition
  - Chapter 7 (7.2), 8(8.2)