Introduction to Object-oriented Programming

Review of structured programming concepts:
Modularity
Cohesion
Top down design

Example: What does this piece of code (pseudocode) do? ("spaghetti code")

```
10 print "headings"
20 goto 50
30 print j
40 goto 200
50 j = 1
60 goto 30
80 j = j + 1
100 if j > 1 then goto 30
200 if j = 100 then stop
210 goto 80
```

same as:

```
print "headings"
do j=1 to 100
print j
```

Go-to less programming:

Why? High-level constructs are clear, concise, efficient, elegant
Exceptions – break, continue, … – use judiciously

any program can be written without the goto statement.
(goto can be either an unconditional branch or a conditional branch)

Even without the built-in high-level syntax, we can write "structured" code:

```
   print "headings"
   j = 1
10 print j
   j = j + 1
   if j <= 100 then goto 10
   …
```

Until now, structured programming referred to control structures governing OPERATIONS. Now we will see how to structure DATA, and then OBJECTS.

Structures

[has-a; is-a]

Fundamental Data Types: char, int, float, etc.
User-defined Data Types: enum, class
Structured Data Types: struct, array, class

A Structure is equivalent to a record.
A record is a collection of data items related to a single object of processing.
A struct is a collection of variables, not necessarily of the same type. These variables are called the members of the structure.

Structures were more important in C. In C++ they are more likely to be part of a class definition. In fact, a struct is almost the same as a class.

A structure enables you to define a new type that logically groups several fields (members) together.

General syntax:

```
struct structName {
     --list of members--
};
```

Examples:

```
struct point {
     double x;
     double y;
};

struct rectangle {
point upperLeftCorner;
point lowerRightCorner;
};

struct circle {
point center;
double radius;
};
```

Defining a structure is like defining a new type. Then you can declare variables of the new type:

```
point p1, p2, p3;
```

or like this:

```
struct point {
     double x;
     double y;
} p1, p2, p3;
```

or even like this:

```
struct {
      double x;
      double y;
} p1, p2, p3;
```

One way to declare and initialize:

```
point p5 = {1.0, -8.3};
```

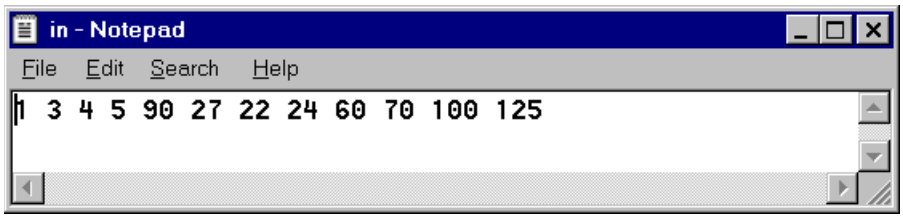
To access individual members, use the dot operator:

```
p1.x = 12.45;
p1.y = 34.56;
p2.x = 23.4 / p1.x;
p2.y = 0.98 * p1.y;
```

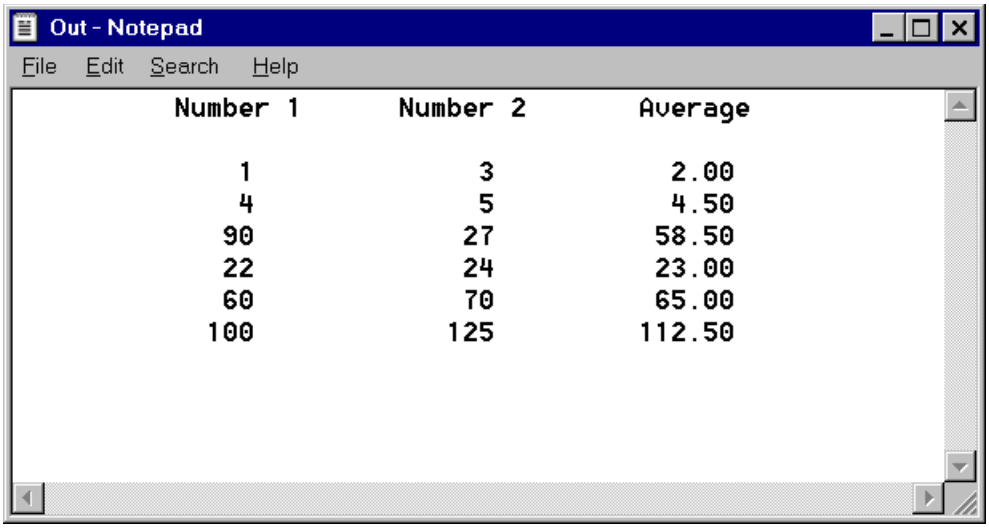
Example:

```
//structs.cpp
//illustration of use of structs for organizing data into records

#include <iomanip>
#include <fstream>
using namespace std;

ifstream infile ("a:in.dat");
ofstream outfile ("a:out.txt");

struct record {                     //struct definition
int num1;
int num2;
};

main(){
record rec;                         //declaring struct variable rec
float avg;

if (!infile)                        //testing files
   cerr << "Error: could not open input file\n";
else if (!outfile)
   cerr << "Error: could not open output file\n";

                                    //printing headings
outfile << setw(18) << "Number 1" << setw(15)<< "Number 2 "
   << setw(15)<< "Average\n\n";

while (infile  >> rec.num1 >> rec.num2){
   avg = (rec.num1 + rec.num2) /2.0;
   outfile << setiosflags(ios::showpoint | ios::fixed)
      << setprecision(2) << setw(15) <<rec.num1 << setw(15) << rec.num2
      << setw(15) << avg << endl;
}

return 0;
}
```

Data File:

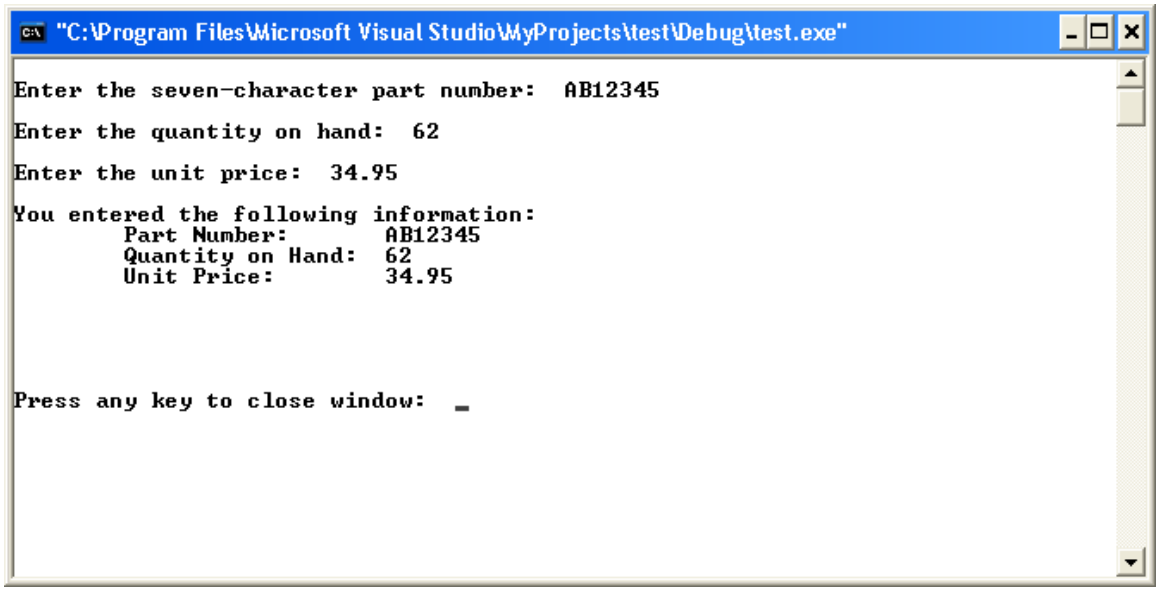


Output file:



Example:

```
//structs1.cpp
//modified from Molluzzo p. 395

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

struct part_struct{
   string part_no;
   int quantity_on_hand;
   double unit_price;
};

int main(){
   part_struct part;
   cout << setprecision(2) << setiosflags(ios::fixed | ios::showpoint);

   cout << "\nEnter the seven-character part number:  ";
   cin >> part.part_no;
   cout << "\nEnter the quantity on hand:  ";
   cin >> part.quantity_on_hand;
   cout << "\nEnter the unit price:  ";
   cin >> part.unit_price;

   cout << "\nYou entered the following information:";
   cout << "\n\tPart Number:       " << part.part_no;
   cout << "\n\tQuantity on Hand:  " << part.quantity_on_hand;
   cout << "\n\tUnit Price:        " << part.unit_price;

   cout << "\n\n\n\n\n\nPress any key to close window:  ";
   char c; cin >> c;
   return 0;
}
```



Example from employee processing program:

```
struct name_struct {
string first_name;
char mid_initial;
string last_name;
}

struct address_struct {
string st_address;
string city;
string state;
string zip;
}

struct employee_struct {
name_struct name;
address_struct address;
string ssnum;
double pay_rate;
}
```

If we declare the following:
`employee_struct employee;`

then, to access part of an employee's file data:
`employee.name.first_name`

Objects and Classes

➔A class is a derived complex data type containing members -- data items and functions that operate on these data -- that may be of different types. A class specifies the traits (data) and behavior that an object can exhibit.

Attributes / member data
Methods / member functions

An attribute is the data defined in a class that maintains the current state of an object. The state of an object is determined by the current contents of all the attributes.

A class itself does not exist; it is merely a description of an object. A class can be considered a template for the creation of object, e.g.,

Blueprint of a building → class
Building → object

QUESTION: Is 'void' a class?

➔An object exists and is definable. An object exhibits behavior, maintains state, and has traits. An object can be manipulated. An object is an instance of a class.

➔A message is a request, sent to an object, that activates a method (i.e., a member function).

➔Inheritance, encapsulation, and polymorphism are the basic principles of object-oriented programming.

Inheritance is the guiding principle for the relationship between classes. This is an explicit is-a relationship. E.g.,

Zebra	is-a	mammal
Flower	is-a	plant
Subclass	→	*superclass*

The more specialized class inherits attributes and behaviors from the more generalized class.

Encapsulation means that a class is cohesive and self-contained.

Polymorphism - poly (*many*) morph (*form*) - if two (or more) objects have the same interface (what the user of the class sees), but exhibit different behaviors, they are said to be polymorphic. Similar to function / operator overloading.

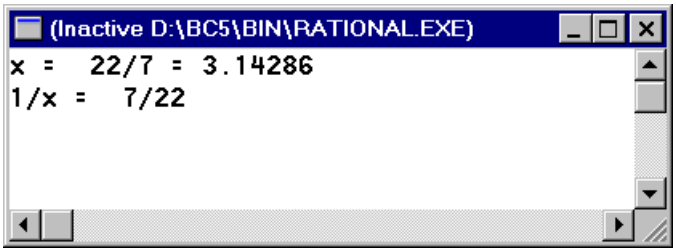
Interface - the visible functionality of a class -- the "what"

Implementation - the internal functionality and attributes of a class. A class's implementation is hidden from users of the class. -- the "how"

Defining a Base Class

A base class is one that is not defined on, nor does it inherit members from, any other class.

```
//fraction.cpp
//modified from Hubbard, Ex. 8.1, p. 221

#include <iostream>
using std::cout;        //this example "uses" only the necessary
using std::endl;        // objects, not the entire std namespace

class Fraction {
public:
   void assign (int, int);                //member functions
   double convert();
   void invert();
   void print();
private:
   int num, den;                          //member data
};

int main(){
   Fraction x;
   x.assign (22, 7);
   cout << "x =  "; x.print();
   cout << " = " << x.convert() << endl;
   x.invert();
   cout << "1/x =  "; x.print(); cout << endl;
   return 0;
}

void Fraction::assign (int numerator, int denominator) {
   num = numerator;
   den = denominator;
}

double Fraction::convert () {
   return double (num)/(den);
}

void Fraction::invert() {
   int temp = num;
   num = den;
   den = temp;
}

void Fraction::print() {
   cout << num << '/' << den;
}
```



The access specifiers provide three levels of visibility. This is how we implement ***information hiding.***

`public` - members are accessible by functions outside the class.
`private` - members are accessible only by functions inside the class. Even descendent classes are denied access.
`protected` - only member functions of the class and its descendents have access.

These can appear in any order. If not specified, the default is `private`.

If a class is kind of like a user-defined type, then, x is an object declared like any other variable. Its type is Fraction.

The prefix `Fraction::` is needed for each member function definition given outside of the class definition.

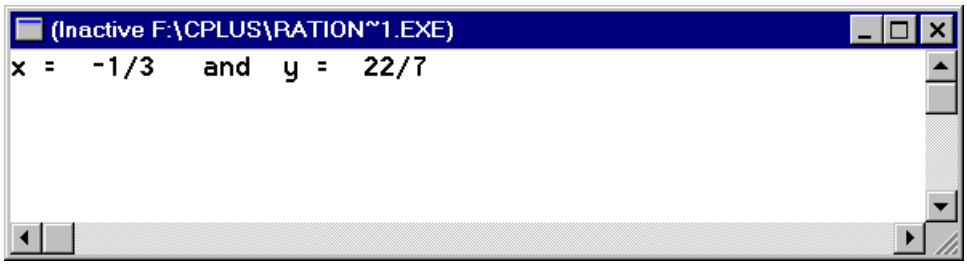
Function definitions can also be included within the declarations inside the class. This is how the print function is defined in the next example. It is not the recommended way of defining class member functions, especially for functions consisting of multiple statements.

Constructors

The assign function in the previous example is an awkward way to initialize objects.

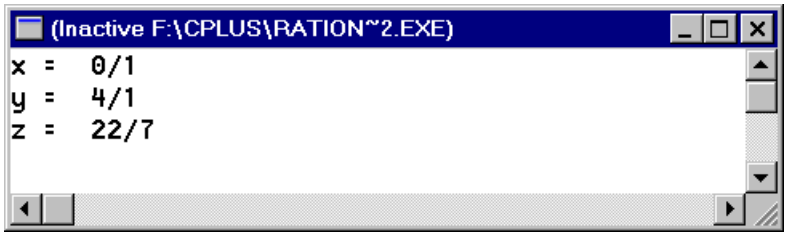
A constructor is a member function that is invoked automatically when an object is declared. It has the same name as the class and carries no return type (not even void). The default constructor, e.g., `Fraction()`, does not take parameters. If the programmer does not supply it, the system will generate one (without default parameter values, of course).

```
//fraction2.cpp
//modified from Hubbard, Ex. 8.3, p. 223

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction (int n, int d) {num = n; den = d;}
   void print(){cout << num << '/' << den;}
private:
   int num, den;
};

int main(){
   Fraction x(-1,3), y(22,7);
   cout << "x =  "; x.print();
   cout << "   and  y =  "; y.print();
   cout << endl;
   return 0;
}
```



Overloading the constructor function:

```
//fraction3.cpp
//modified from Hubbard, Ex. 8.4, p. 224

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction() {num = 0; den = 1;}            //default parameter values
   Fraction(int n) {num = n; den = 1;}       //only 1 default value
   Fraction (int n, int d) {num = n; den = d;}
   void print(){cout << num << '/' << den;}
private:
   int num, den;
};

int main(){
   Fraction x, y(4), z(22,7);
   cout << "x =  "; x.print();
   cout << "\ny =  "; y.print();
   cout << "\nz =  "; z.print();
   cout << endl;
   return 0;
}
```



Which constructor is called? That depends on the parameter list in the declaration of the object.

Using constructor initialization lists. The following program will produce same output as the one above:

```
//fraction4.cpp
//modified from Hubbard, Ex. 8.5, p. 225

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction() : num (0), den (1) {}
   Fraction(int n) : num (n), den (1) {}
   Fraction (int n, int d) : num (n), den (d) {}
   void print(){cout << num << '/' << den;}
private:
   int num, den;
};

int main(){
   Fraction x, y(4), z(22,7);
   cout << "x =  "; x.print();
   cout << "\ny =  "; y.print();
   cout << "\nz =  "; z.print();
   cout << endl;
   return 0;
}
```

Also this:

```
//fraction5.cpp
//modified from Hubbard, Ex. 8.6, p. 226

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction (int n=0, int d=1) : num (n), den (d) {}
   void print(){cout << num << '/' << den;}
private:
   int num, den;
};

int main(){
   Fraction x, y(4), z(22,7);
   cout << "x =  "; x.print();
   cout << "\ny =  "; y.print();
   cout << "\nz =  "; z.print();
   cout << endl;
   return 0;
}
```

When actual parameters are not passed, the default values are used.

Constant Objects and Constant Member Functions

Objects may be declared to be constant, eg.

```
const Fraction PI(22,7);
```

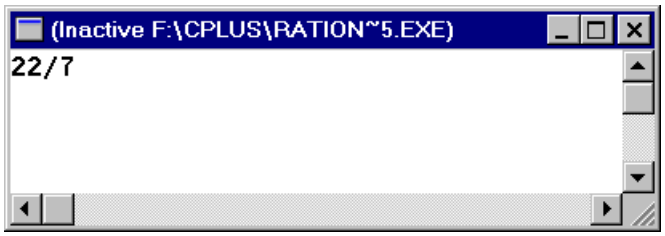
This will limit access to the object's member functions. For ex, the print function could not be called for PI.print(). To enable access of the objects member functions when the objects is declared a constant, we must declare the member functions const as well, e.g.,

```
void print() const {cout <<num << '/' << den << endl;}
```

Access Functions – 'get' functions

It is common practice to include public functions that provide a "read-out" of the values of an object's private data items. These *access functions*, also called *get functions*, will frequently be labeled `const` functions so that they cannot change the values of the object data, only report them.

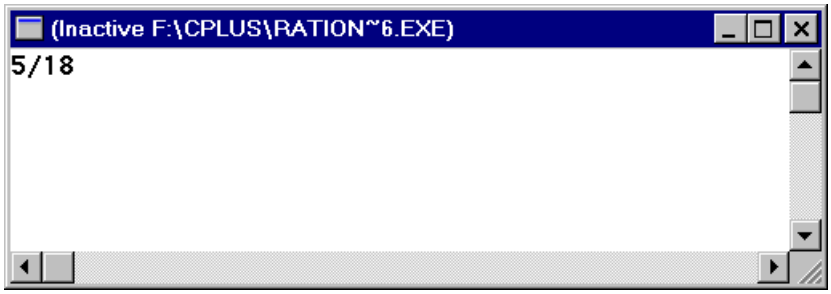
```
//fraction6.cpp
//modified from Hubbard, Ex. 8.7, p. 226

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction (int n=0, int d=1) : num (n), den (d) {}
   int getnumerator() const {return num;}
   int getdenominator() const {return den;}
private:
   int num, den;
};

int main(){
   Fraction x(22,7);
   cout << x.getnumerator() <<'/'
      << x.getdenominator() << endl;
   return 0;
}
```


(Inactive F:\CPLUS\RATION~5.EXE)
22/7

Private Member Functions

Sometimes we need private "utility" functions, used only by the object itself. No outsider needs access to them.

```
//fraction7.cpp
//modified from Hubbard, Ex. 8.8, p. 227

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction (int n=0, int d=1) : num (n), den (d) {reduce();}
   void print(){cout << num << '/' << den;}
private:
   int num, den;
   int gcd (int j, int k) {if (k==0) return j; return gcd(k, j%k);}
   void reduce () {int g = gcd(num, den); num /= g; den /= g;}
};

int main(){
   Fraction x(100,360);
   x.print();
   return 0;
}
```



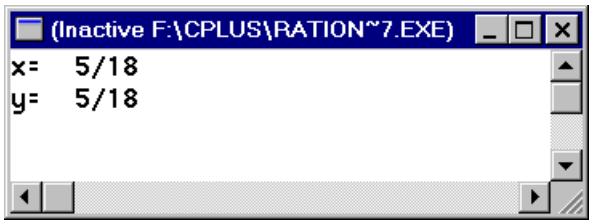
Copy Constructor

The copy constructor builds an object by copying the state of an existing object into a new object of the same class. There are two default constructors; if necessary they will be automatically provided by the system. These are, e.g.,

```
Fraction ();  and
Fraction (const Fraction&);
```

It's better to include it ourselves. This gives us more control over the program.

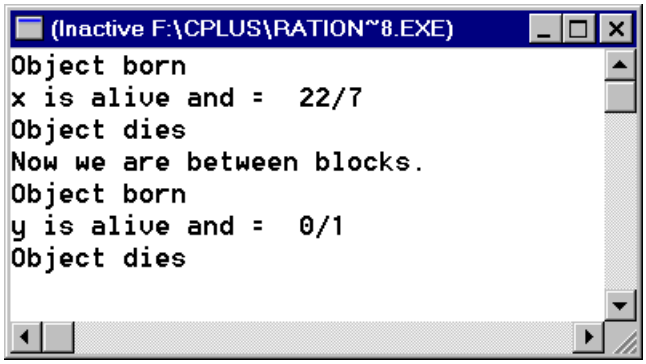
```
//fraction8.cpp
//modified from Hubbard, Ex. 8.9, p. 228

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction (int n=0, int d=1) : num (n), den (d) {reduce();}
   Fraction (const Fraction& r) : num(r.num), den(r.den){}
   void print(){cout << num << '/' << den;}
private:
   int num, den;
   int gcd (int j, int k) {if (k==0) return j; return gcd(k, j%k);}
   void reduce () {int g = gcd(num, den); num /= g; den /= g;}
};

int main(){
   Fraction x(100,360);
   Fraction y(x);
   cout << "x=  "; x.print();
   cout << "\ny=  "; y.print();
   return 0;
}
```



Destructor

Just like a constructor is called automatically when an object is declared/created, so a destructor is called automatically when an object comes to the end of its life. The destructor has the same name as the class, except that the name must begin with a tilde (~). It has no return type, not even void. If not defined explicitly, a default destructor will be created by the system. A class may have multiple constructors, but only one destructor. The destructor is not called explicitly.

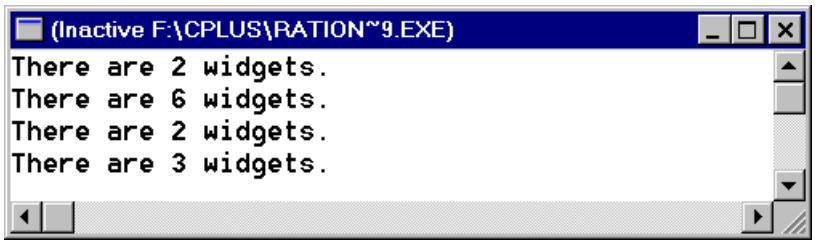
```
//fraction9.cpp
//modified from Hubbard, Ex. 8.11, p. 230

#include <iostream>
using namespace std;

class Fraction {
public:
   Fraction (int n=0, int d=1) : num (n), den (d)
      {reduce(); cout << "Object born";}
   ~Fraction() {cout << "\nObject dies";}            //destructor
   void print(){cout << num << '/' << den;}
private:
   int num, den;
   int gcd (int j, int k) {if (k==0) return j; return gcd(k, j%k);}
   void reduce () {int g = gcd(num, den); num /= g; den /= g;}
};

int main(){
   {  Fraction x(22,7);
      cout << "\nx is alive and =  "; x.print();
   }

   cout << "\nNow we are between blocks.\n";

   {  Fraction y;
      cout << "\ny is alive and =  "; y.print();
   }

   return 0;
}
```



Static Data Members

Static attributes are used when a single value for a data member applies to all objects of the class. The variable must be declared globally. These variables are automatically initialized to 0.

```
//fraction10.cpp
//modified from Hubbard, ex. 8.15, p. 235

#include <iostream>
using namespace std;

class Widget {
public:
   Widget() {++count;}
   ~Widget() {--count;}
   int numWidgets() {return count;} //access function
private:
   static int count;                //private and static
};                                  //accessible to all objects of the
class

int Widget:: count = 0;

int main(){
   Widget w, x;
   cout << "There are " << w.numWidgets() << " widgets.\n";
   {
      Widget w,x,y,z;
      cout << "There are " << w.numWidgets() << " widgets.\n";
   }
   cout << "There are " << w.numWidgets() << " widgets.\n";
   Widget y;
   cout << "There are " << w.numWidgets() << " widgets.\n";
   return 0;
}
```



A function can also be static. Since a static function is not "owned" by any single object of the class, it can be called even before any objects are declared.

Improved Program:

```
//fraction11.cpp
//modified from Hubbard, ex. 8.16, p. 236

#include <iostream>
using namespace std;

class Widget {
public:
   Widget() {++count;}
   ~Widget() {--count;}
   static int num() {return count;}
private:
   static int count;
};

int Widget::count = 0;

int main(){
   cout << "There are " << Widget::num() << " widgets.\n";
   Widget w, x;
   cout << "There are " << Widget::num() << " widgets.\n";
   {
      Widget w,x,y,z;
      cout << "There are " << Widget::num() << " widgets.\n";
   }
   cout << "There are " << Widget::num() << " widgets.\n";
   Widget y;
   cout << "There are " << Widget::num() << " widgets.\n";
   return 0;
}
```

Exercise

(a) Implement a Student class. Each object of this class will represent a student in a particular course. Data members should include the student's ID number, first name, last name, and four exam scores. Include a constructor, access functions, a displayStudentInfo function, an inputStudentInfo function, and a computeAvgScore function.

In main(), create an object of this class and use it to enter values for some of its data members from the keyboard and display the values of its data members in the console window. You may ask the user to enter the number of students in the class. Use a loop to enter data, compute the average, and output data for successive students.

Use the following test data:
100 Robin Williams 90 85 67 99
200 Duncan McLeod 35 67 88 100
300 Bart Simpson 67 44 89 67

(b) Modify the program in (a), above, to compute and display the value of the overall average score for the class

More Examples:

(from Hubbard, p. 245)
Implement a Computer class with data members for the computer type (e.g., "pc"), the CPU (e.g., "Intel Pentium"), the operating system (e.g., "DOS"), the number of megabytes of memory (e.g., 16), the number of gigabytes of disk space (e.g., 9.8), the type of printer attached, type of CD_ROM drive, type and speed of modem, internet service provider, purchased price, and year of purchase. Include a default constructor, a destructor, access functions, and a print function.

(modified from Hubbard, p. 245)
Implement a Student class. Each object of this class represents a student. Data members should include a student's name (first name, last name, middle initial), date of birth (month, day, year), student identification number, major program, grade point average, and credits earned. Include a default constructor, a default destructor, access functions, and print function. Also include a member function `update (int course, credit, char grade)` that processes the given information (*course*, *credit* and *grade*) for one course, using it to update the student's grade point average and credits earned.

(modified from Molluzzo, p. 489)
Implement a HotelRoom class, with private data members: the room number, room capacity (representing the maximum number of people the room can accommodate), the occupancy status (0 or the number of occupants in the room), the daily room rate. Member functions include:

- a 4-argument constructor that initializes the four data members of the object being created (room number, room capacity, room rate, occupancy status) to the constructor's arguments. The room rate defaults to 89.00 and the occupancy status defaults to 0.
- A destructor
- accessor functions
- a print function
- a function changeRate that sets the room rate to the value of its argument
- a function changeStatus that changes the occupancy status of the room to the value of its argument. The function should verify that the argument value does not exceed the room capacity; if it does, the function should return -1.

Write a main() that creates a hotel room with room number 123, a capacity of 4, and a rate of 150.00. Suppose a person checks in. The program should ask the user to enter the number of guests to occupy the room. Change the status of the room to reflect the number of guests that just checked in. Display the information about the room in a nice format. Now assume that the guests check out. Change the status of the room appropriately and display the information about the room. Next, change the room rate to 175.00. Finally assume that another person checks in. Ask the user to enter the number of guests to occupy the room. Change the room's status accordingly and display the new information about the room.