Classes: Methods, Constructors, Destructors and Assignment

For : COP 3330.
Object oriented Programming (Using C++)
http://www.compgeom.com/~piyush/teach/3330

Piyush Kumar

Next week's homework

- Read Chapter 7
- Your next quiz will be on Chapter 7 of the textbook

Classes: Member functions

```
// classes example
#include <iostream>

class Square {
        int x;
  public:
        int area () {return (x*x);};
        void set_values(int a);
        int get_sidelen(void) const;
};

// Typically in a .cpp file
void Square::set_values(int a){x = a;}
int Square::get_sidelen(void) const { return x;}
```

Member functions

```
int main () {
  Square s;
  s.set_values(3);
  std::cout << "area: "
            << s.area();
  return 0;
}

// Output: "area: 9"
```

Member functions: Methods

- Private members of a class are accessible only from within other members of the same class or from their *friends*.
- Protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

Objects : Reminder

- An object is an instance of a class.
- Memory is allocated for each object instantiated (and not for the class).
  - Example:
    - Square S; // S is an object of Square class. (on stack)
    - Square *pS = new Square; // on heap.
- An object of a class can be defined in the same way as an internal type.

Objects : Reminder

- Multiple objects of a class can be created (as many as you want).
- All the objects share the same copy of member functions.
- But, they maintain a separate copy of data members.
  - Square s1,s2; // each has separate copy of x

Objects: Reminder

- The data members and member functions of an object have the same properties as the data members and member functions of its class.

Further study:
https://www.programiz.com/cpp-programming/object-class

Assignment operator

- Square s3 = s2;
- *By default, copying a class object is equivalent to copying all its elements including pointers.*

Variable assignment

- Values are assigned to variables and not to their data types.
- Hence, you assign values to members of an object and not a class.
- Must create a unique object of a class because you cannot assign values to a class.
  - Square = 5 is meaningless…

Back to member functions

- The member functions in the public section of a class can be accessed using the "." operator for instantiated objects. (For pointers its ->)
  - Square s1, *s2;
  - s1.set_values(5);
  - s2 = new Square; s2->set_values(10);
  - delete s2;
- Only the public members of an object can be directly accessed.

Special Member functions

- Constructors: Are invoked to initialize the data members of a class.
- Can not return any value (not even void).
- Can accept any parameters as needed.
  - What would happen if we called the member function area() before having called function set_values()?

Constructors vs. Destructors

- Constructors initialize the objects in your class.
- Destructors clean up and free memory you may have allocated when the object was created.

Constructors

- Differs from other member functions.
- Initializes a newly created object.
- Other member functions are invoked by existing objects.
- A Constructor is invoked automatically when an object is created.

Constructors

- Have the same name as the class.

```
#include <iostream>

class Square {
      int x;
  public:
      Square(int w){ x = w; };
      int area () {return (x*x);};
      void set_values(int a);
      int get_sidelen(void) const;
};
void Square::set_values(int a){ x = a; }
int Square::get_sidelen(void) const { return x;}
```

```
int main () {
      Square s(3);
      std::cout << "area: " << s.area();
      return 0;
}
// Output: area: 9
```

Constructors

- Have the same name as the class.

```
#include <iostream>

class Square {
      int x;
  public:
      Square(int w){ x = w; };
      int area () {return (x*x);};
      void set_values(int a);
      int get_sidelen(void) const;
};
void Square::set_values(int a){ x = a; }
int Square::get_sidelen(void) const { return x;}
```

```
int main () {
      Square s;
      cout << "area: " << s.area();
      return 0;
}

// Output: area: 0
```

Default constructor is defined for you: Square(){};

Constructors: Overloading

- You can have several constructors for a class by overloading them.

```
class Square {
      int x;
  public:
      Square(int w){ x = w; };
      Square() { x = 0; };
      int area () {return (x*x);};
      void set_values(int a);
      int get_sidelen(void) const;
};
```

Constructors : Initialization

- Prefer *member initializer list* to assignment
  - Square(int w):x(w){};
  - Array(int lowbound, int highbound):size(highbound-lowbound+1), lb (lowbound), hb (highbound), data_vector(size) {};
  - Problematic? Why?
- List members in an initialization list in the order they were declared in the class

Constructors: Warning

- If you implement no constructor, the compiler automatically generates a default constructor for you
- But if you write any constructors at all, the compiler does not supply a default constructor.

Copy Constructors

- Gets called in a number of situations.
- If you do not write one, the compiler automatically generates one for you.

Copy constructors: When is it called?

- When the return value of a function has class type.
  - Fraction process_fraction (int i, int j);
- When an argument has class type. A copy of the argument is made and passed to the function
  - int numerator_process (Fraction f);
- When you use one object to initialize another object.
  - Fraction a(1,3); Fraction b(a);
  - Fraction b = a;

Copy constructors

- Not used when a pointer to an object is passed.
- It's only called when a new copy of an existing object needs to be created.

Copy constructors

- The syntax:
  - `class_name(class_name const &source)`
- Const: Making a copy should not alter source.
- &: The function should not need to call another copy constructor!

Copy Constructors: Example.

```
class Point {
        int x,y;

public:
        int xx(void) const { return x;};
        int yy(void) const { return y;};
        Point(Point const &p){
                this->x = p.xx();
                this->y = p.yy();
        };
};
```

Destructors

- You can have many constructors but only one destructor.
- The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.
- The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// source: cplusplus.com
#include <iostream>

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area () {return (*width * (*height));}
};

CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a; *height = b;
}

CRectangle::~CRectangle () { delete width; delete height; }

int main () {
  CRectangle rect (3,4), rectb (5,6);
  std::cout << "rect area: " << rect.area() << std::endl;
  std::cout << "rectb area: " << rectb.area() << std::endl;
  return 0;
}
```

Output:

"rect area: 12
rectb area: 30"

this pointer

- Useful when a member function manipulates two or more objects.
- It holds the address of the object for which the member function is invoked.
- It is always passed to a non-static member function. This ensures the right object is updated using member functions.

Back to destructor example

- What will happen if we do:
  - CRectangle r1(10,20),r2(30,40);
  - r1 = r2;
  - The default assignment operator generated by the compiler is called. What is wrong with that in this example?

Assignment operators

- int x;
- x = 4;
- 4 = x; // non-lvalue in assignment
- x = y = z = 8;
  - z is assigned 8 first
  - y is then assigned the value returned by (z = 8) which is 8.
  - x is now assigned the value returned by (y = 8) which is 8

Assignment operators

- Complex x, y, z;
- x = (y = z);
- Writing this in equivalent functional form:
  - x.operator=(y.operator=(z));

Assignment operators

- Square s1 = s2;
- The default behavior : Performs simple member by member copy.

(This is the one generated by the compiler)

Is the assignment operator the same thing as copy constructor?

Assignment Operator

- Copy constructor initializes an object.
- Assignment operator copies values to an existing object.
- Hence, in some cases: Copy constructor has to do more work than assignment operator.

Assignment Operator

- Syntax:
  - `class_name& operator=(const class_name &source)`
- const: Making an assignment should not alter source.
- &: The function should not need to call a copy constructor.

An Example

```
#include <iostream>
#include <cstring>

class Buf{
  public:
    Buf( char const* szBuffer, std::size_t sizeOfBuffer );
    Buf& operator=( const Buf & );
    void Display() { std::cout << buffer << std::endl; }

  private:
    char*   buffer;
    std::size_t  SizeOfBuffer;
};
```

```
Buf::Buf( char const* szBuffer, std::size_t sizeOfBuffer )
{
   sizeOfBuffer++; // account for a NULL terminator

   buffer = new char[ sizeOfBuffer ];
   if (buffer)
   {
      memcpy( buffer, szBuffer,  sizeOfBuffer );
      SizeOfBuffer = sizeOfBuffer;
   }
}

Buf& Buf::operator=( const Buf &otherbuf )
{
   if( &otherbuf != this )
   {
      if (buffer)
         delete [] buffer;

      SizeOfBuffer =  strlen( otherbuf.buffer ) + 1;
      buffer = new char[SizeOfBuffer];
      memcpy( buffer, otherbuf.buffer, SizeOfBuffer );
   }
   return *this;
}
```

```
int main()
{
   Buf myBuf( "my buffer", 10 );
   Buf yourBuf( "your buffer", 12 );

   // Display 'my buffer'
   myBuf.Display();

   // assignment opperator
   myBuf = yourBuf;

   // Display 'your buffer'
   myBuf.Display();

   return 0;
}
```

Output:

"my buffer
your buffer"

Guidelines

- *Place the common code used by the assignment operator and the copy constructor in a separate function and have each one call the function. This will make your code more compact and avoid duplication.*
- A String class must copy a character string during the copy constructor and during an assignment. If we place this common code into a private member function
- void CopyString(const char* ptr); then both the copy constructor and assignment operator may call this routine to perform the copy rather than duplicate this code in each.

From: http://www.acm.org/crossroads/xrds1-4/ovp.html

Guidelines

- *If your class has pointer data, you **must** provide an assignment operator. If writing an assignment operator, you must also write a copy constructor (and destructor?).*
- The generated assignment operator performs member-wise assignment on any data members of your class. For pointer variables, we almost always do not want this because the data members of the copy will point to the same data as the copied object! Worse, if one of the objects is destroyed, the data is destroyed with it. A run-time error will occur the next time the remaining object tries to access the now non-existent data.

Guidelines

- *When dealing with pointers, Always implement the assignment operator for your class; do not let the compiler generate the default assignment operator. (Remember rule of 3)*
- The compiler will generate a default assignment operator for your class if you do not provide one. In order to be in complete control of how your class operates, always provide an assignment operator.

Guidelines

- *Check for assignment to self.*
- Disaster can result if a variable is assigned to itself. Consider:
- X x; x = x; Suppose class X contains pointer data members that are dynamically allocated whenever an object is created. Assignment always modifies an existing object. The dynamic data will, therefore, be released before assigning the new value. If we do not check for assignment to self, the above assignment will delete the existing pointer data, and then try to copy the data that was previously deleted!

Guidelines

- *The destructor must release any resources obtained by an object during its **lifetime**, not just those that were obtained during construction.*
- *Make the constructor as compact as possible to reduce overhead and to minimize errors during construction.*