CSC212

Data Structure

COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

Lecture 3
ADT and C++ Classes (II)

Instructor: George Wolberg
Department of Computer Science
City College of New York

Outline

A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

Standard Library & Namespace

- ANSI/ISO C++ Standard (late 1990s)
  - aids in writing portable code with different compilers
- C++ Standard Library (1999 C++ compilers provide full SL)
  - Provides a group of declared constants, data types and functions, such as I/O and math
  - Use new "include directive" such as #include <iostream> without .h
- Standard Namespace
  - All the items in the new header files are part of a feature called standard namespace std
  - When you use one of the new header files, you should use

    using namespace std
  - which allows you to use all items from the standard namespace.

Namespace and Documentation

- Goal:
  - to make our new point class easily available to any programs any time without
    - revealing all the details
    - worrying about name conflicts
- Three steps to fulfill the goal
  - Creating a namespace
  - Writing the header file
  - Writing the implementation file

```
Namespace wolberg_ccny_csc212_lecture_3
{
    // any item that belongs to the namespace is written here
}
```

- Question:
  - You may use two versions of point classes in the same program
- Solution is to use the namespace technique
  - A namespace is a name that a programmer selects to identify a portion of his/her work
  - The name should be descriptive, better include part of your real name and other features for uniqueness

Namespace groupings

- All work that is part of our namespace must be in a namespace grouping
- A single namespace such as wolberg_ccny_csc212_lecture_3 may have several namespace groupings
- They don't need in the same files, typically in two separate files
  - Class definition in a header file
  - Member function definitions in a separate implementation file

Header File for a Class

- A separate header file for a new class
  - point.h
- At the top place the documentation (how to use)
- Followed by class definition (but not the implementation)
- Place class definition inside a namespace
- Place a "macro guard" around the entire thing
- Documentation should include a comment indicating that the value semantics is safe to use

Implementation File for a Class

- A separate implementation file for a new class
  - point.cpp (or point.cxx, point.C)
- At the top place a small comment indicating the documentation is in the header file
- Followed by include directive #include “point.h”
- reopen the namespace and place the implementation of member functions inside the namespace

Using Items in a Namespace

- A separate program file for using classes
  pointmain1.cpp
- At the top place an include directive
  #include “point.h”
- Three ways to use the items in a namespace
  - using namespace main_savitch_2A;
  - using main_savitch_2A::point;
  - main_savitch_2A::point p1;
- Question: shall we include the implementation file in pointmain1.cpp?

Outline

A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

Classes and Parameters

- Default parameters
  - when no or only part of the parameters are provided in calling function
- Types of parameters
  - value parameters
  - reference parameters
  - constant reference parameters
- Return value is a class

Default arguments

- A default argument is a value that will be used for an argument when a programmer does not provide an actual argument when calling a function
- Default arguments may be listed in the prototype of a function
  - Syntax: Type_name var_name = default_value

Default arguments – rules

- The default argument is only specified once – in the prototype – not in the implementation

Example of a prototype:

```
int date_check (int year, int month = 1, int date =1);
```

- In a call, arguments with default may be omitted from the right end.

Default arguments – rules

Example:

```
int date_check (int year, int month = 1, int date =1); // okay

int date_check (int year =2002, int month = 1, int date); // ?
```

- No need to specify all the arguments as default but those as default must be the rightmost in the parameter list
- In a call, arguments with default may be omitted from the right end.

Default arguments – rules

Prototype:

```
int date_check (int year, int month = 1, int date =1);
```

Usage in the calling function

```
date_check(2002); // uses default  for both month and date

date_check(2002, 9); // uses default  for date =1

date_check(2002, 9, 5); // does not use defaults
```

- In a call, arguments with default may be omitted from the right end.

How can we apply default arguments to a constructor ?

Default Constructor revisited

- A default constructor can be provided by using default arguments

```
class point
{
public:
    point();
    point(double x, double y);
    …
};
```

- Instead of define two constructors and have two implementations

Default Constructor revisited

- A default constructor can be provided by using default arguments

```
class point
{
public:
    point(double x=0.0, double y =0.0);
     …
};
```

- We can define just one constructor with default arguments for all of its arguments

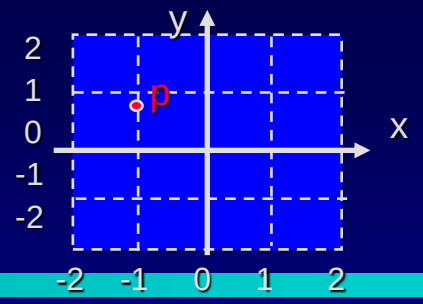
Default Constructor revisited

- In using the class, we can have three declarations

```
point a(-1, 0.8);  // uses the usual constructor with
                   // two arguments
point b(-1); // uses –1 for the first,
             // but use default for the second

point c; // uses default arguments for both;
         // default constructor:
         // no argument, no parentheses!
```

- The implementation of the constructor with default argument is the same as the usual one...

Constructors: Implementation



And for the most part, the constructor is no different than any other member functions.

```
point::point(double x, double y)
{
    m_x = x;
    m_y = y;
}
```

But recall that there are 3 special features about constructors...and 4 for this with default arguments!

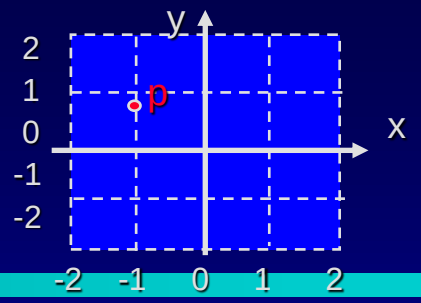
Second topic about parameters....

Classes as parameters

Class as type of parameter

- A class can be used as the type of a function's parameter, just like any other data type
  - Value parameters
  - Reference parameters
  - Const reference parameters
  - In fact you can also have const value parameters, even if this does not make much sense

Value parameters



- How many shifts to move p into the first quad

Function implementation:

```
int shifts_needed(point p)
{
    int answer = 0;
    while ((p.x() <0) || (p.y()<0))
    {
        p.shift(1,1);
        answer++;
    }
    return answer;
}
```

formal parameter

actual argument

-1.5, -2.5

3

-1.5, -2.5

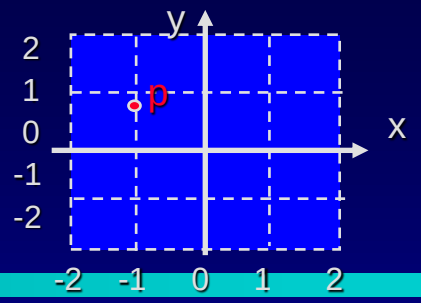
In calling program:

```
point a(-1.5,-2.5);
cout << a.x() << a.y() << endl;
cout << shifts_needed(a) << endl;
cout << a.x() << a.y() << endl;
```

Value parameters

- A value parameter is declared by writing
  - type-name parameter-name
- Any change made to the formal parameter within the body of the function does not change the actual argument from the calling program. This is call-by-value.
- The formal parameter is implemented as a local variable of the function, and the class's copy constructor is used to initialize the formal parameter as a copy of the actual argument

Reference parameters



- Actually move p into the first quadrant

Function implementation (almost the

type_name & para_name

```
int shift_to_1st_quad(point& p)
{
    int shifts;
    while ((p.x() <0) || (p.y()<0))
    {
        p.shift(1,1);
        shifts++;
    }
    return shifts;
}
```

NO & !

-1.5, -2.5

3

1.5, 0.5

In calling program:

```
point a(-1.5,-2.5);
cout << a.x() << a.y() << endl;
cout << shift_to_1st_quad(a) << endl;
cout << a.x() << a.y() << endl;
```

Reference parameters

- A reference parameter is declared by writing
  - type-name& parameter-name
- Any use of the formal parameter within the body of the function will access the actual argument from the calling program; change made to the parameter in the body of the function will alter the argument. This is called call-by-reference.
- The formal parameter is merely another name of the argument used in the body of the function!

const reference parameters

- A const reference parameter is declared by writing
  - const type-name& parameter-name
- A solution that provides the efficiency of a reference parameter along with the security of a value parameter.
- Example (newpoint.cpp)
  - double distance (const point& p1, const point& p2)
    - point p1 and p2 cannot be changed (TEST!)

Third topic about parameters and functions of a class....

Class as return value

Class as return value

```
point middle(const point& p1, const point& p2)
{
        double x_midpoint, y_midpoint;

        // compute the x and y midpoints
        x_midpoint = (p1.x( ) + p2.x( )) / 2;
        y_midpoint = (p1.y( ) + p2.y( )) / 2;

        // construct a new point and return it
        point midpoint(x_midpoint, y_midpoint);
        return midpoint;
}
```

Class as return value

- The type of a function's return value may be a class
- Often the return value will be stored in a local variable of the function (such as midpoint), but not always (could be in a formal parameter)
- C++ return statement uses the copy constructor to copy the function's return value to a temporary location before returning the value to the calling program
- Example (Ch 2.4, Look into newpoint.cpp)
  - point middle(const point& p1, const point& p2)

Outline

A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

Operator Overloading

- Binary functions and binary operators
- Overloading arithmetic operations
- Overloading binary comparison operations
- Overloading input/output functions
- Friend functions – when to use

Operator Overloading

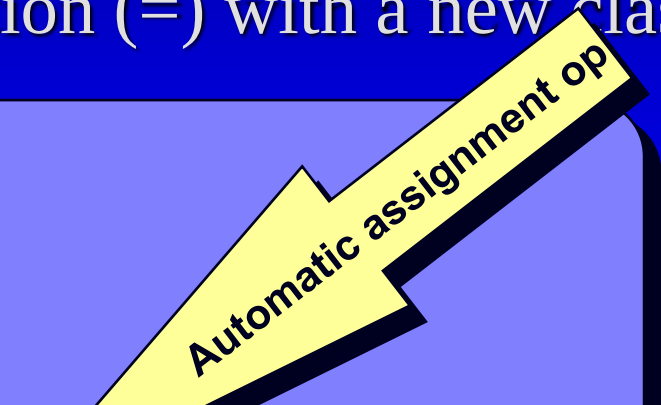
- Question:
  - Can we perform arithmetic operations (+ - * /) or comparison operations (>, ==, <, etc.) or assignment operation (=) with a new class?

```
point speed1(5,7)
point speed2(1,2);
point difference;

if (speed1 != speed2 )
        difference = speed1 - speed2;
```

Operator Overloading

- Question:
  - Can we perform arithmetic operations (+ - * /) or comparison operations (>, ==, <, etc.) or assignment operation (=) with a new class?

```
point speed1(5,7)
point speed2(1,2);
point difference;

if (speed1 != speed2 )
        difference = speed1 - speed2;
```



Operator Overloading

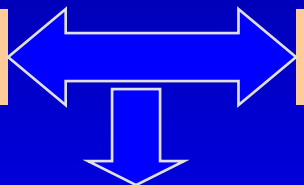
- Answer is NO
  - unless you define a binary function that tells exactly what "!=" or "+" means

```
point speed1(5,7)
point speed2(1,2);
point difference;

if (speed1 != speed2 )
        difference = speed1 - speed2;
```

Operator Overloading

- Binary Function
  - A function with two arguments
- Binary Operator
  - A operator with two operands

```
p = add( p1, p2);
```



```
p = p1 + p2;
```

Operator Overloading is to define the meaning of an existing operator for a new class

Instead of defining

```
point add(point p1, point p2)
```

We define

```
point operator+(point p1, point p2)
```

Overloading arithmetic operators

- +, -, *, / , %

```
point operator+(const point& p1, const point& p2)
//Postcondition: the sum of p1 and p2 is returned.
{
   double x_sum, y_sum;
   x_sum = (p1.x() + p2.x());
   y_sum = (p1.y() + p2.y());
   point sum(x_sum, y_sum);
   return sum;
}
```

Overloading arithmetic operators

- Apart from the peculiar name operator+, the function is just like any other function
- The overloaded operator + is used in a program just like any other use of +
  - p = p1+ p2;
- When you overload an operator +, the usual usage of + is still available
- Note the uses of
  - const reference parameters since…
  - member functions x and y instead of variables
  - the function is a nonmember function

Overloading arithmetic operators

- Method 1: Nonmember function p = p1+p2

```
point operator+(const point& p1, const point& p2)
//Postcondition: the sum of p1 and p2 is returned.
{
   double x_sum, y_sum;
   x_sum = (p1.x() + p2.x());
   y_sum = (p1.y() + p2.y());
   point sum(x_sum, y_sum);
   return sum;
}
```

Overloading arithmetic operators

- Method 2: Member function p = p1+p2

```
point point::operator+(const point& p2) const
//Postcondition: the sum of activating object (p1) and
argument p2 is returned.
{
    double x_sum, y_sum;

    x_sum = (m_x + p2.x());

    y_sum = (m_y + p2.y());

    point sum(x_sum, y_sum);

    return sum;

}
```

Overloading arithmetic operators

- Overloading using nonmember function
  - PROs: two arguments on equal footing
  - CONs: cannot use the member variables
- Alternative ways to overload a binary function
  - Member function
    - PROs: can use member variables
    - CONs: p1 activate the operator with argument p2
- Which way do you prefer?

Overloading comparison operators

- ==, !=, <, >, <=, >=

```
bool operator==(const point& p1, const point& p2)

//Postcondition: the return is true if p1 and p2 are identical;
otherwise return is false.

{

   return( (p1.x() == p2.x()) && (p1.y() == p2.y()) );

}
```

Overloading comparison operators

- ==, !=, <, >, <=, >=

```
bool operator!=(const point& p1, const point& p2)

//Postcondition: the return is true if p1 and p2 are NOT
identical; otherwise return is false.

{

   return ( (p1.x() != p2.x()) || (p1.y() != p2.y()) );

}
```

Overloading comparison operators

- ==, !=, <, >, <=, >=

```
bool operator!=(const point& p1, const point& p2)

//Postcondition: the return is true if p1 and p2 are NOT
identical; otherwise return is false.

{

   return !(p1== p2);

}
```

- Or use the overloaded operator for easy implementation

Overloading I/O operators

- Input (>>) & Output (<<) for a new class: <<

```
ostream& operator<<(ostream& outs, const point& source)
//  Postcondition: The x and y coordinates of source have been
//  written to outs. The return value is the ostream outs.
//  Library facilities used: iostream
    {
          outs << source.x( ) <<  " "  << source.y( );
          return outs;
    }
```

- Q1: how to use this overloaded operator?

```
cout << p ;
```

Overloading I/O operators

- Input (>>) & Output (<<) for a new class: <<

```
ostream& operator<<(ostream& outs, const point& source)
//  Postcondition: The x and y coordinates of source have been
//  written to outs. The return value is the ostream outs.
//  Library facilities used: iostream
    {
        outs << source.x( ) <<  " "  << source.y( );
        return outs;
    }
```

- Q2: why is outs a reference parameter but NOT const?

Need change actual argument cout

Overloading I/O operators

- Input (>>) & Output (<<) for a new class: <<

```
ostream& operator<<(ostream& outs, const point& source)
//  Postcondition: The x and y coordinates of source have been
//  written to outs. The return value is the ostream outs.
//  Library facilities used: iostream
    {
        outs << source.x( ) <<  " "  << source.y( );
        return outs;
    }
```

- Q3: why return ostream&?

For chaining: cout << “The point is” << p << endl;

Overloading I/O operators

- Input (>>) & Output (<<) for a new class: <<

```
ostream& operator<<(ostream& outs, const point& source)
//  Postcondition: The x and y coordinates of source have been
//  written to outs. The return value is the ostream outs.
//  Library facilities used: iostream
    {
        outs << source.x( ) <<  " "  << source.y( );
        return outs;
    }
```

- Q4: How to overload the input operator >> ?

Overloading I/O operators

- Input (>>) & Output (<<) for a new class: >>

```
istream& operator>>(istream& ins, point& target)
//  Postcondition: The x and y coordinates of target have been
//  read from ins. The return value is the istream ins.
//  Library facilities used: iostream
    {
        ins >> target. x >> target.y;
        return ins;
    }
```

- NO const for both istream and point
- Problem: send input directly to private members!

Three possible solutions

- Use a member function for overloading the input function (try!)
- Write new member functions to set a point's coordinates separately so they can be used within the input function (try!)
- Grant special permission for the input function to access the private variables
  - using a friend function

Friend Function

- A friend function is NOT a member function, but it still has access to the private members of its parameters

```
class point
{
public:
    … …
    // FRIEND FUNCTION
    friend istream& operator>>(istream& ins, point& target);
private:
    …
};
```

Overloading I/O operators

- Input (>>) & Output (<<) for a new class: >>

```
istream& operator>>(istream& ins, point& target)
//  Postcondition: The x and y coordinates of target have been
//  read from ins. The return value is the istream ins.
//  Library facilities used: iostream
    {
          ins >> target. x >> target.y;
          return ins;
    }
```

- Problem is resolved by using friend function, no change in implementation

Overloading I/O operators

- Input (>>) & Output (<<) for a new class: >>

```
istream& operator>>(istream& ins, point& target)
//  Postcondition: The x and y coordinates of target have been
//  read from ins. The return value is the istream ins.
//  Library facilities used: iostream
//  Friend of point class
    {
          ins >> target. x >> target.y;
          return ins;
    }
```

- However it is always a good practice to put a comment line

Summary of Classes

A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

point class: Putting things together

- Header file (newpoint.h)
  - Documentation including pre- & post-conditions
  - Class definitions for any new classes //inline
  - Prototype of nonmember functions (e,g. for overloading)
  - Place the Class and Prototype inside a namespace
- Implementation file (newpoint.cpp)
  - An include directive to include the header file
  - Implementation of each function (except inline)
  - Implementation of each friend and other nonmember
  - Use the same namespace for implementation
- Calling program file (pointmain2.cpp)
  - Three ways to use the items in a namespace

Exercises and Assignments

- Writing Homework
  - Alternative implementation of operator >>
- Self-Test Exercises (do not turn in)
  - 1, 4 , 5, 13,15,17,21,23, 25,28,31
- Reading before the next lecture
  - Chapter 3. Container Classes
- Programming Assignment 1
  - Detailed guidelines online!
    - check schedule on our course web page

END