COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

Classes and Abstract Data Types

Slide Set: 2

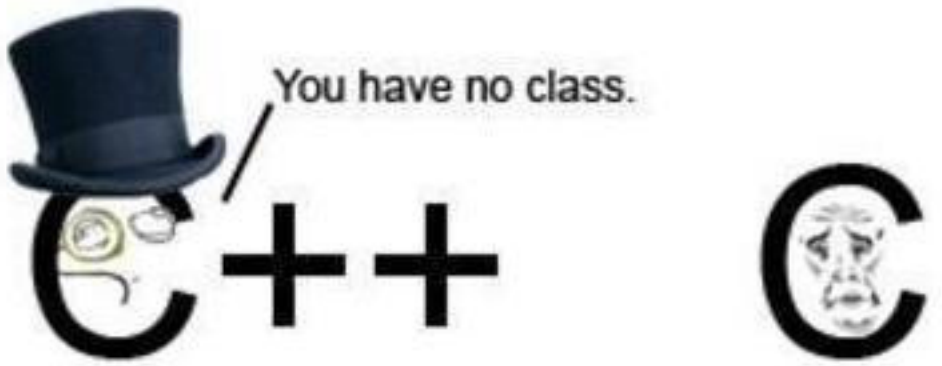
Learning Objectives

- Specification and design of new classes using a pattern of **information hiding** with private member variables, const member functions, and modification member functions
- Write a header file and a separate implementation file for any new class
- Create and use **namespaces** to organize new classes
- Use your new classes (and at least one STL class) in small test programs
- Use the **automatic assignment operator** and the **automatic copy constructor**
- How member functions and constructors can benefit from using default arguments
- Correctly identify and use **value parameters, reference parameters**, and **const reference parameters**
- Overload certain **binary operators** and input/output operators for new classes
- Identify the need for **friend functions** of a new class and correctly implement such **nonmember functions** (which are sometimes overloaded operators)
- Use STL classes, such as the pair class, in an application program

Introduction

Introduction

- **Object-oriented programming (OOP)** is an approach to programming in which data occurs in tidy packages called **objects**
- Manipulation of an object happens with functions called **member functions**, which are part and parcel of their objects
- Programmers sometimes use the term **Abstract Data Type (ADT)** to refer to a class that is presented to other programmers with information hiding


You have no class.
C++
C

Classes and Members

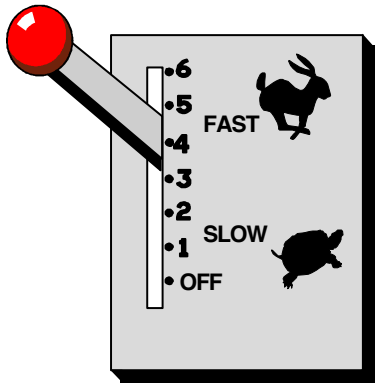
Classes and Members

- **A class is a new kind of data type**
  - A collection of data, such as integers, characters, and so on
  - We implement data structures as classes
  - A class has the ability to include special functions, called **member function** which are designed specifically to manipulate the class

  - Note: Data type = Data structure = Container

Example: The Throttle Class

- The **Throttle** class is a new data type to store and manipulate the status of a simple throttle
- A lever that can be moved to control fuel flow
- A single shutoff point (where there is no fuel flow) and a sequence of six on positions where the fuel is flowing at progressively higher rates
- For example, with six possible positions, and the lever in the fourth position, the fuel flows at 4/6 of its maximum rate



Example: The Throttle Class (Cont'd)

- Functions provided:
  - A function to set a throttle to its shutoff position
  - A function to shift a throttle's position by a given amount
  - A function that returns the fuel flow, expressed as a proportion of the maximum flow
  - A function to tell us whether the throttle is currently on

- The new **data type** as a **"class"** called **throttle** that includes:
  - Data (to store the throttle's current position)
  - Four functions to modify and examine the throttle

Example: The Throttle Class (Cont'd)

```
class throttle
{

public:

    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( );
    void shift(int amount);


    // CONSTANT MEMBER FUNCTIONS
    double flow( ) const;
    bool is_on( ) const;


private:
    int position;

};
```

Class head consists of C++ keyword `class` and name of the class

Function to set a throttle to its shutoff position

Function to shift a throttle's position by a given amount

- Function that returns the fuel flow
- Constant member functions' prototypes have the keyword `const` at the end

Function to tell us whether the throttle is currently on

Member variable of the class to store the throttle's current position

Classes and Members

- A Common Pattern for Classes:
  - **Public member functions** permit programmers to modify and examine objects of the new class
    - Use the keyword **`const`** (after the function's parameter list) when a member function examines data without making modifications
  - **Private member variables** of the class store the information about the status of an object of the class

- Note:
  - The class body contains prototypes for the member functions but not the full definitions of these functions

Using a Class

- You may declare throttle variables
  - These variables are called throttle **objects**, or sometimes throttle **instances**

  ❑Example:
    - `throttle my_throttle;`
    - `throttle control;`

- The only way that a program can use its throttle objects is by using the four public member functions

❑Example: we want to set `control` to its third notch

- `control.shut_off( );` (you can pronounce "control dot shut off")
- `control.shift(3);`

We activated the `shut_off` and `shift` member functions

Using a Class (Cont'd)

- Class vs. Object
  - **A class** is a kind of data type that defines data members and member functions that operate on the data
  - **An object is an instance of a class**, and is declared as a variable
  - Once a class is defined, a programmer can declare many objects of that class and manipulate the objects with its functions

Classes and Members

❑Example

```
…
throttle control;
int user_input;
control.shut_off( );

cout << "Please type a number from 0 to 6: ";
cin >> user_input;
control.shift(user_input);

if (control.is_on( ))
        cout << "The flow is " << control.flow( ) << endl;
else
        cout << "The flow is now off" << endl;
…
```

- **We don't worry about how the member functions accomplish their work**
- **We simply activate each member function and wait for it to return**

A Small Demonstration Program for the Throttle Class

```
// This small demonstration shows how the throttle class is used.

#include <iostream>  // Provides cout and cin
#include <cstdlib>   // Provides EXIT_SUCCESS
using namespace std; // Allows all Standard Library items to be used

class throttle
{

public:
    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( );
    void shift(int amount);

    // CONSTANT MEMBER FUNCTIONS
    double flow( ) const;
    bool is_on( ) const;

private:
    int position;
};
```

A Small Demonstration Program for the Throttle Class (Cont'd)

```
int main( )
{
    throttle sample;
    int user_input;

    cout << "I have a throttle with 6 positions." << endl;
    cout << "Where would you like to set the throttle? " << endl;
    cout << "Please type a number from 0 to 6: ";
    cin >> user_input;
    sample.shut_off( );
    sample.shift(user_input);

    while (sample.is_on( ))
    {
        cout << "The flow is now " << sample.flow( ) << endl;
        sample.shift(-1);
    }

    cout << "The flow is now off" << endl;
    return EXIT_SUCCESS;
}
```

Implementing Member Functions

Writing definitions for member functions is just like writing any other function

- **One small difference: In the head of the function definition, the class name must appear before the function name, separated by two colons** (Why we need this rule?)
- Referred to as the **scope resolution protocol**

```
void throttle::shut_off( )
// Precondition: None.
// Postcondition: The throttle has been turned off.
{
    position = 0;
}
```

- We use the term **function implementation** to describe a full function definition

Implementing Member Functions (Cont'd)

Member Variables

- **Each object keeps its own copies of all member variables**
- When a member function's implementation refers to a member variable, then **the actual member variable used always comes from the object that activated the member function**

```
throttle big;
throttle low;

big.shut_off( );
low.shut_off( );
big.shift(6);
low.shift(1);

cout << "The big flow is: " << big.flow( ) << endl;
cout << "The low flow is: " << low.flow( ) << endl;
```

Declare two throttles.

Set the positions of the throttle's levers.

Print the flows.

- The first output statement prints 1.0 (which is `big`'s flow (6/6))
- The second output statement prints 0.166667 (which is `low`'s flow (1/6))

Implementing Member Functions (Cont'd)

- Why the shift function has a precondition indicating that "`shut_off` has been called at least once to initialize the throttle"?

```
void throttle::shift(int amount)
// Precondition: shut_off has been called at least once to initialize
// the throttle.
// Postcondition: The throttle's position has been moved by amount
// (but not below 0 or above 6).
{
    position += amount;
    if (position < 0)
        position = 0;
    else if (position > 6)
        position = 6;
}
```

Implementing Member Functions (Cont'd)

```
double throttle::flow( ) const
// Precondition: shut_off has been called at least once to initialize
// the throttle.
// Postcondition: The value returned is the current flow as a
// proportion of the maximum flow.
{
    return position / 6.0;
}
```

```
bool throttle::is_on( ) const
// Precondition: shut_off has been called at least once to initialize
// the throttle.
// Postcondition: If the throttle's flow is above 0, then the function
// returns true; otherwise, it returns false.
{
    return (flow( ) > 0);
}
```

Our implementation of `is_on` activates the `flow` member function

Constructors

- **Constructors** provide an initialization function **that is guaranteed to be called**
- **A constructor is a member function** with these special properties:
  - A constructor is **called automatically** whenever a variable of the class is declared
  - The **name of a constructor must be the same as the name of the class**
  - A constructor **does not have any return value**
    - You must not write `void` (or any other return type) at the front of the constructor's head

The Throttle's Constructor

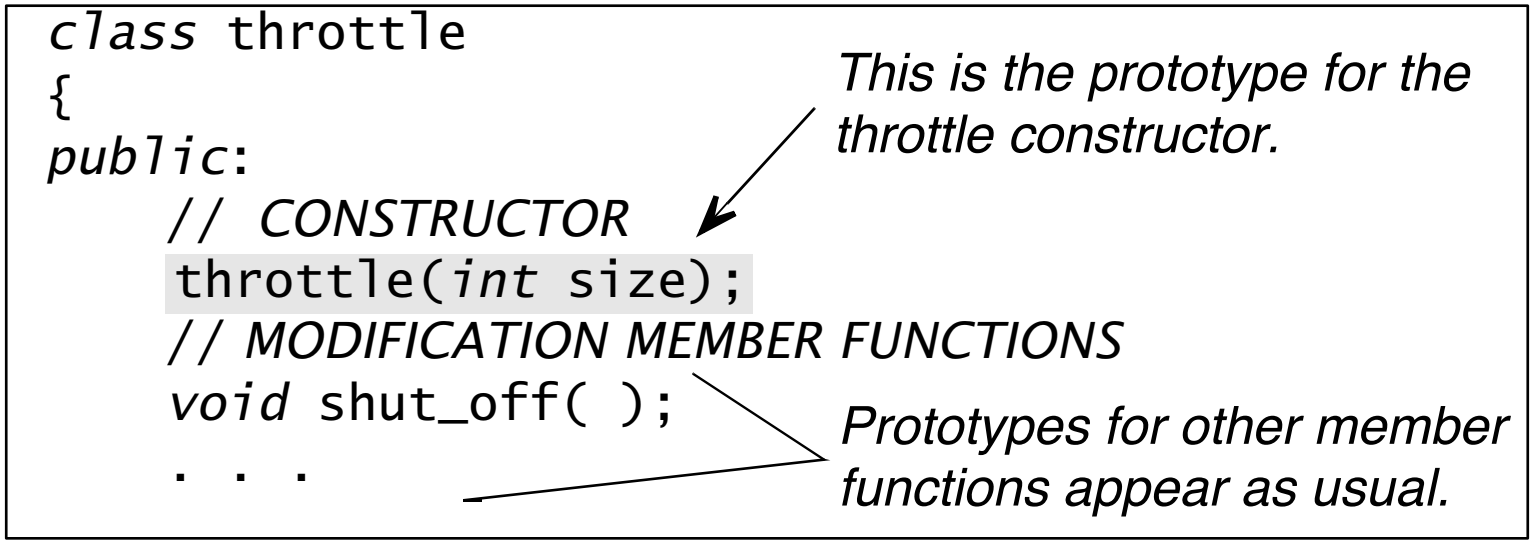
Our defined constructor will actually make the throttle more flexible by **allowing the total number of throttle positions to vary from one throttle to another**

- Has one parameter, which tells the total number of positions that the throttle contains

```
throttle(int size);
// Precondition: 0 < size.
// Postcondition: The throttle has size positions above the shutoff position,
// and its current position is off.
```

Note: The word `void` does not appear at the front of the prototype, nor is there any other return type for the function

The Throttle's Constructor (Cont'd)



- Example: Declaration of two throttles:
  - `throttle mower_control(3);` (has 3 positions)
  - `throttle apollo(7);` (has 7 positions)

The Throttle's Constructor (Cont'd)

It is **useful to provide several different constructors**, each of which does a different kind of initialization

- Suppose many of our throttles require just one on position

```
throttle( );
// Precondition: None.
// Postcondition: The throttle has one position above the shutoff position,
// and its current position is off.
```

A constructor with no parameters is called a **default constructor**

❑ **Example:**

```
throttle toggle;
```

When toggle uses the default constructor, there is no argument list (not even a pair of parentheses)

Classes and Members

The Throttle's Constructor (Cont'd)

```
class throttle
{
public:

    // CONSTRUCTORS
    throttle( );
    throttle(int size);

    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( );
    void shift(int amount);

    // CONSTANT MEMBER FUNCTIONS
    double flow( ) const;
    bool is_on( ) const;

private:
    int top_position;
    int position;
};
```

```
throttle::throttle( )
{
    top_position = 1;
    position = 0;
}
```

```
throttle::throttle(int size)
// Library facilities used: cassert
{
    assert(0 < size);
    top_position = size;
    position = 0;
}
```

A new private member variable keeps track of how many positions the throttle has

Constructors

If you write a class with no constructor:

- The compiler automatically creates a simple default constructor
- **This automatic default constructor doesn't do much work**

Inline Member Functions

- **Placing a function definition inside the class definition is called an inline member function**

It has two effects:

- You don't have to write the implementation later
- Each time the inline function is used in your program, **the compiler will recompile the short function definition and place a copy of this compiled short definition in your code**
  - This **saves some execution time** (no actual function call and return)
  - **May be inefficient** in space (many copies of the same compiled code)
- **Use an inline member function only for the simple situation when the function definition consists of a single short statement**

- We use this technique to place the complete definitions of `shut_off`, `flow`, and `is_on` **inside the class definition**

Inline Member Functions (Cont'd)

```
class throttle
    {

    public:
        // CONSTRUCTORS
        throttle( );
        throttle(int size);

        // MODIFICATION MEMBER FUNCTIONS
        void shut_off( ) { position = 0; }
        void shift(int amount);

        // CONSTANT MEMBER FUNCTIONS
        double flow( ) const
                { return position / double(top_position); }
        bool is_on( ) const { return (position > 0); }

    private:
        int top_position;
        int position;
    };
```

The highlighted code shows three inline member functions

The type name "double" changes `top_position` from an integer to a double number
The change is called a "**type cast**," and it prevents an unintended integer division

Using a Namespace, Header File, and an Implementation File

Using a Namespace, Header File, and an Implementation File

- Goals:
  - To make our new throttle class easily available to any program that needs it, **without revealing all the details** of the new class's implementation
  - Other programmers **should not worry about** whether their own selection of names for variables and such will **conflict** with the names that we happen to use

To this end:

- Create a **namespace**
- Write the **header file**
- Write the **implementation file**

Creating a Namespace

- **Namespaces** provide a method **for preventing name conflicts** in large projects
- A **namespace** is a name that a programmer selects to identify a portion of his or her work

```
namespace scu_coen79_2A
{
  ||Any item that belongs to the namespace is written here.
}
```

- A single namespace, such as `scu_coen79_2A`, may have several different namespace groupings
- For example: one for class definition, one for implementation

Creating a Namespace (Cont'd)

The **Global Namespace:**

- Any items that are not explicitly placed in a namespace become part of the global namespace
  - **These items can be used at any point without any need for a `using` statement or a scope resolution operator**

C++ Standard Library (`std` namespace):

- **All of the items in the C++ Standard Library are automatically part of the `std` namespace**
  - When using C++ header file names (such as `<iostream>` or `<cstdlib>`)
  - The simplest way is to add "`using namespace std;`"

Creating a Namespace (Cont'd) – An Example

```
#include <vector>
namespace vec
{
    template< typename T >
    class vector
    {
        // ...
    };
} // of vec

int main()
{
    std::vector<int> v1;

    vec::vector<int> v2;
    ...
}
```

C++ std's vector

User-define vector

- Show how name conflict happens if we do not use a namespace when we define a new data structure

The Header File

- The class definition appears in a **header file**
  - Provides all the information that a programmer needs in order to use the class
  - All the **information needed to use the class** should appear in **a header file comment** at the top of the header file

- The member function definitions appear in a separate **implementation file**

The Header File (Cont'd)

- When a program includes a header file more than once, **compilation fails** because of "**duplicate class definition**"
- Avoid duplicate class definition by placing all the header file's definitions inside a compiler directive called a **macro guard**

```
#ifndef scu_coen79_THROTTLE_H
#define scu_coen79_THROTTLE_H
namespace SCU_coen79_2A
{
    class throttle
    {
        ||class definition
    };
}
#endif
```

These statements will be compiled only if the compiler has not yet seen a definition of the word **SCU_coen79_THROTTLE_H**

The Header File (Cont'd)

❑Example

- What is the problem?

File "a.h"

```
class foo
{
    …
};
```

File "b.h"

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

File "c.c"

```
#include "a.h"
#include "b.h"
```

What is the solution?

Describing the Value Semantics of a Class Within the Header File

- The **value semantics** of a class determines **how values are copied from one object to another**
- In C++, the value semantics consists of two operations:
  - **Assignment operator**
  - **Copy constructor**

Describing the Value Semantics of a Class Within the Header File (Cont'd)

❖The assignment operator

❑Example:

- DS1 is a data structure that includes the names of employees in a hospital
- DS2 is an object that can include names of employees
- We want to assign the values in DS1 to DS2

This is what we do:

```
employeeNames DS2;
DS2 = DS1;
```

- The **automatic assignment operator** simply **copies each member variable** from the object on the right of the assignment to the object on the left of the assignment
- Note: Automatic assignment operator **does not work always** (we will see in future chapters)

Describing the Value Semantics of a Class Within the Header File (Cont'd)

❖The copy constructor

- **A constructor with exactly one argument**
- The data type of the argument is the same as the constructor's class

❑Example:

- DS1 is a data structure that includes the names of employees in a hospital
- We want to create an object DS2 while making it an exact copy of DS1

This is what we do:

```
employeeNames DS2(DS1);
```

- **An alternative syntax**: `employeeNames DS2 = DS1;`
  - Declares a new object, `DS2`
  - **Calls the copy constructor** to initialize `DS2` as a copy of `DS1`

Describing the Value Semantics of a Class Within the Header File (Cont'd)

- C++ provides an **automatic copy constructor**
- The automatic copy constructor initializes a new object by **merely copying all the member variables from the existing object**

- When you implement a class, **the documentation should include a comment indicating that the value semantics is safe to use**

```
//  VALUE SEMANTICS  for the throttle class:
//       Assignments and the copy constructor may be used with throttle
//       objects.
```

Implementation File

- **Implementation File** has several items:
  1. A small comment appears, indicating that the documentation is available in the header file
  2. An include directive appears, causing the compiler to grab the class definition from the header file
     - Example: `#include "throttle.h"`

Note:
- We use quotation marks rather than angle brackets
- The angle brackets (e.g., `#include <iostream>`) are used only to include a Standard Library facility

  3. The program reopens the namespace and gives the implementations of the class's member functions

Implementation File (Cont'd)

```
// FILE: throttle.cxx
// CLASS IMPLEMENTED: throttle (See throttle.h for documentation.)
```

```
#include <cassert>     // Provides assert
#include "throttle.h"  // Provides the throttle class definition
```

```
using namespace std;   // Allows all Standard Library items to be used

namespace scu_coen79_2A
{
        ||functions implemented
}
```

Using the Items in the Namespace

- When header and implementation files are ready:
  - Any program can use our new class
  - At the top of the program, place an `include` directive to include the header file
    - Example: `#include "throttle.h"`

Using the Items in the Namespace (Cont'd)

- How to use the items that are defined in the namespace:
  1. **Make all of the namespace available**

     Syntax: `using namespace ns_name;`

     ❑ Example: `using namespace scu_coen79_2A;`

     - Question: Why is this a bad idea to put a using statement in a header file?

  2. If we need to **use only a specific item from the namespace:**

     Syntax: `using ns_name::name;`

     ❑Example: `using scu_coen79_2A::throttle;`

  3. Use any item by **prefixing the item name with the namespace and "::"** at the point where the item is used

     Syntax: `ns_name::name`

     ❑Example: `scu_coen79_2A::throttle apollo;`

Using the Items in the Namespace (Cont'd)

A Sample Program

```
#include <iostream>
#include <cstdlib>
#include "throttle.h"

using namespace std;
using scu_coen79_2A::throttle;

// Number of positions in a demonstration throttle
const int DEMO_SIZE = 5;


int main( )
{
    throttle  sample(DEMO_SIZE);
    cout << "I have a throttle with " << DEMO_SIZE << " positions."
                                                    << endl;
```

Using the Items in the Namespace (Cont'd)

A Sample Program (Cont'd)

```
    cout << "Where would you like to set the throttle?" << endl;
    cout << "Please type a number from 0 to " << DEMO_SIZE << ": ";

    cin >> user_input;
    sample.shift(user_input);

    while (sample.is_on( ))
    {
        cout << "The flow is now " << sample.flow( ) << endl;
        sample.shift(-1);
    }

    cout << "The flow is now off“ << endl;
    return EXIT_SUCCESS;
}
```

Classes and Parameters

Classes and Parameters

- Classes can be used as the type of a **function's parameter**, or as the **type of the return value** from a function

❑For example:

- You may pass a data structure to a function to apply some changes to the data structure
- The return value of a function may be a data structure

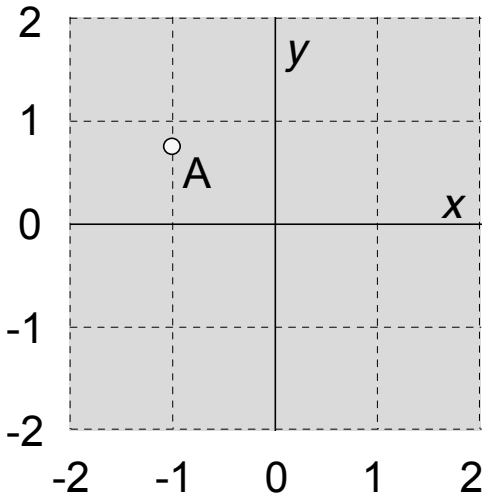
Programming Example: The Point Class

- The **`point`** class: Is a data structure to store and manipulate the location of a single point on a plane

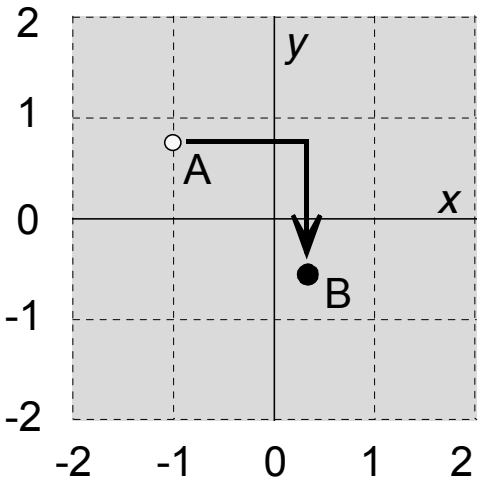
- The point class has the member functions listed here:
  - There is a constructor to initialize a point
  - There is a member function to shift a point by given amounts along the x and y axes
  - There is a member function to rotate a point by 90° in a clockwise direction around the origin
  - There are two constant member functions that allow us to retrieve the current x and y coordinates of a point

Programming Example: The Point Class (Cont'd)

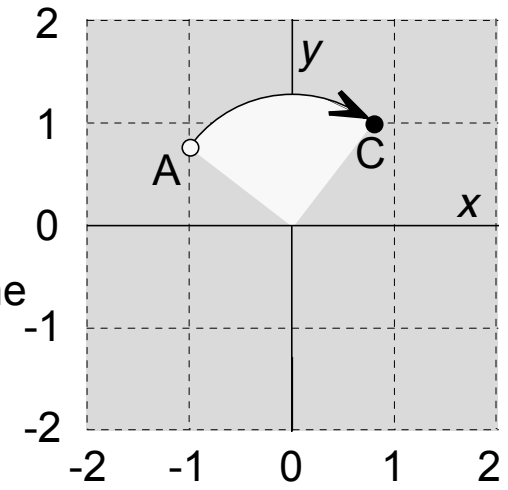
(a) The white dot labeled A is a point with coordinates $x = -1.0$ and $y = 0.8$.



(b) The black dot labeled B was obtained by shifting point A by 1.3 units along the x axis and by -1.4 units along the y axis. The coordinates of point B are $x = 0.3$ and $y = -0.6$.



(c) The black dot labeled C was obtained by rotating point A 90° in a clockwise direction around the origin. The coordinates of point C are $x = 0.8$ and $y = 1.0$.



Programming Example: The Point Class (Cont'd)

Header file of the Point class

```
namespace scu_coen79_2A
{
    class point
    {
    public:
        // CONSTRUCTOR
        point(double initial_x = 0.0, double initial_y = 0.0);
        // MODIFICATION MEMBER FUNCTIONS
        void shift(double x_amount, double y_amount);
        void rotate90( );
        // CONSTANT MEMBER FUNCTIONS
        double get_x( ) const { return x; }
        double get_y( ) const { return y; }
    private:
        double x; // x coordinate of this point
        double y; // y coordinate of this point
    };
}
```

Default Arguments (discussed later)

Programming Example: The Point Class (Cont'd)

Implementation file of the Point class

```
#include "point.h"

namespace scu_coen79_2A
{
    point::point(double initial_x, double initial_y)
    {
        x = initial_x;
        y = initial_y;
    }


    void point::shift(double x_amount, double y_amount)
    {
        x += x_amount;
        y += y_amount;
    }
```

Programming Example: The Point Class (Cont'd)

Implementation file of the Point class (Cont'd)

```
    void point::rotate90( )
    {
        double new_x;
        double new_y;

        // For a 90 degree clockwise rotation, the new x is the
        // original y,
        new_x = y;
        // and the new y is -1 times the original x.
        new_y = -x;
        x = new_x;
        y = new_y;
    }
}
```

Default Arguments

- **Default argument** is a value that will be used for an argument when a programmer does not provide an actual argument

❑ Example:

- `point(double initial_x = 0.0, double initial_y = 0.0);`
  (Look at the header file of the point class)

- `point a(-1, 0.8);`

Uses the usual constructor with two arguments

- `point b(-1);`

Uses -1 for the first argument and uses the default argument, `initial_y = 0.0`, for the second argument

- `point c;`

Uses default arguments for both `initial_x = 0.0` and `initial_y = 0.0`

Default Arguments (Cont'd)

- The default argument is specified only once (in the prototype) and not in the function's implementation
- A function with several arguments does not need to specify default arguments for every argument
- If only some of the arguments have defaults, **then those arguments must be right-most in the parameter list**
- In a function call, arguments with default values **may be omitted from the right end** of the actual argument list

❑ Example:

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

- `date_check(2000);`
- `date_check(2000, 7);`
- `date_check(2000, 7, 22);`

Parameters

- Three different kinds of **parameters**:
  - **Value Parameters**
  - **Reference Parameters**
  - **Const Reference Parameters**

Parameters (Cont'd)

❖ **Value Parameters**

❑ Example:

The function's parameter (p in this case) is referred to as the **formal parameter** to distinguish it from the value that is passed in during the function call

```
int rotations_needed(point p)
// Postcondition: The value returned is the number of 90-degree
// clockwise rotations needed to move p into the upper-right
// quadrant (where x >= 0 and y >= 0).
{
    int answer;
    answer = 0;
    while ((p.get_x( ) < 0) || (p.get_y( ) < 0))
    {
        p.rotate90( );
        ++answer;
    }
    return answer;
}
```

Parameters (Cont'd)

❖ Value Parameters

❑ Example (Cont'd):

```
point sample(6, -4); // Constructor places the point at x = 6, y = –4.
cout << " x coordinate is " << sample.get_x( ) << " y coordinate is "
                            << sample.get_y( ) << endl;

cout << " Rotations: " << rotations_needed(sample) << endl;

cout << " x coordinate is " << sample.get_x( ) << " y coordinate is "
                            << sample.get_y( ) << endl;
```

The **passed value** (`sample` in this case) is the **argument** (sometimes called the **actual argument** or the **actual parameter**)

```
Output:
x coordinate is 6 y coordinate is -4
Rotations: 3
x coordinate is ?  y coordinate is  ?
```

What is the output?

Parameters (Cont'd)

❖Value Parameters

- A **value parameter** is declared by writing the type name followed by the parameter name
- With a value parameter, the **argument** provides the initial value for the **formal parameter**
- The value parameter is implemented as a **local variable** of the function
  - Therefore: Any changes made to the parameter in the body of the function **will leave the argument unaltered**

Parameters (Cont'd)

❖Reference Parameters

❑ Example:

```
void rotate_to_upper_right(point& p)
// Postcondition: The point p has been rotated in 90-degree increments
// until p has been moved into the upper-right quadrant
// (where x >= 0 and y >= 0).
{
    while ((p.get_x( ) < 0) || (p.get_y( ) < 0))
        p.rotate90( );
}
```

Parameters (Cont'd)

❖ Reference Parameters

❑ Example:

```
point sample(6, -4); // Constructor places point at x = 6, y = –4.

cout << " x coordinate is " << sample.get_x( ) << " y coordinate is "
                            << sample.get_y( ) << endl;

rotate_to_upper_right(sample);

cout << " x coordinate is " << sample.get_x( ) << " y coordinate is "
                            << sample.get_y( ) << endl;
```

```
Output:
x coordinate is 6 y coordinate is -4
x coordinate is 4 y coordinate is 6
```

Parameters (Cont'd)

❖Reference Parameters

- A **reference parameter** is declared by writing the type name followed by the character & and the parameter name
- With a reference parameter, any use of the parameter within the body of the function will access the argument from the calling program
- Changes made to the formal parameter in the body of the function **will alter the argument**

Parameters (Cont'd)

☞Pitfall

- In order for a reference parameter to work correctly, **the data type of an argument must match exactly with the data type of the formal parameter**

```
void make_int_2(int& i)
// Postcondition: i has been set to 2.
{
    i = 2;
}
```

```
double d;
d = 0;
make_int_2(d);
cout << d;
```

Does not change d
Because d is the wrong data type, a separate *integer* copy of d is created to use as the argument

```
Output:
0
```

Parameters (Cont'd)

If the argument's data type does not exactly match the data type of the formal parameter:

- The compiler will try to **convert** the argument to the correct type
- If the conversion is possible, then **the compiler treats the argument like a value parameter**, passing a **copy** of the argument to the function

Parameters (Cont'd)

❖Const Reference Parameters

- When we don't want a programmer to worry about whether the function changes the actual argument
- A solution that provides **the efficiency of a reference parameter** along with the **security of a value parameter**

❑Example:

- A function that computes the distance between two points
- The function has two point parameters, and **neither parameter is changed by the function**

```
double distance( const point& p1, const point& p2 );
```

When the Type of a Function's Return Value is a Class

- The type of a function's return value may be a class

```
point middle(const point& p1, const point& p2)
// Postcondition: The value returned is the point that
// is halfway between p1 and p2.
{
    double x_midpoint, y_midpoint;

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

    // Construct a new point and return it.
    point midpoint(x_midpoint, y_midpoint);

    return midpoint;
}
```

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

Operator Overloading

Operator Overloading

- A **binary function** is a function with two arguments
- When we develop a new data structure, we usually need to overload the binary operators

❑For example: For the point class, **we cannot use the regular == operator to decide if two points are equal**

- We need to **overload** the == operator to compare particular member variables of the two objects

```
point p1, p2;
if (p1 == p2)
    cout << "Those points are equal." << endl;
```

Binary Operators That Are Often Overloaded As Comparison Functions

==	!=
<	>
<=	>=

- Defining a new meaning for an operator is called **overloading the operator**

Overloading Binary Comparison Operators

- The name of the new function is **`operator ==`**

```
bool operator == (const point& p1, const point& p2)
// Postcondition: The value returned is true if p1 and p2
// are identical; otherwise false is returned.
{
    return
        (p1.get_x( ) == p2.get_x( )) &&
        (p1.get_y( ) == p2.get_y( ));
}
```

Note:

- When overloading an operator, **the common usages of that operator are still available**
- For example, we can use == to test the equality of two integers or two doubles
- **For each use of ==, the compiler determines the data type of the objects being compared and uses the appropriate comparison function**

Overloading Binary Arithmetic Operators

❑Example

If we could add two of our points, then we might write this program

```
point speed1(5, 7);
point speed2(1, 2);
point total;
total = speed1 + speed2;
cout << total.get_x( ) << endl;
cout << total.get_y( ) << endl;
```

```
Output:
6
9
```

Binary Operators That Are Often Overloaded As Arithmetic Functions

+	-
*	/
%	

Overloading Binary Arithmetic Operators (Cont'd)

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

    // Compute the x and y of the sum.
    x_sum = (p1.get_x( ) + p2.get_x( ));
    y_sum = (p1.get_y( ) + p2.get_y( ));

    point  sum( x_sum, y_sum );

    return sum;
}
```

Overloading Output and Input Operators

- The standard C++ data types can be written and read using the **output operator <<** and **the input operator >>**

```
int i;
cin >> i;
cout << i;
```

reads the value of i from the standard input

writes the value of i to the standard output

- We would like to support the following input and output operations for the point class

```
point p;
cin >> p;
cout << p;
```

reads the x and y coordinates of p from the standard input

writes the x and y coordinates of p to the standard output

Overloading Output and Input Operators (Cont'd)

- Overloading the output operator for the `point` class

```
ostream& operator <<(ostream& outs, const point& source);
```

- The data type of `cout` is `ostream`, which means "output stream"
- `ostream` class is part of the `iostream` library facility

- The `outs` parameter is a reference parameter
- The function can change the output stream (by writing to it), and the change will affect the actual argument (such as the standard output stream, `cout`)

❑Example

```
ostream& operator <<(ostream& outs, const point& source);
```

```
cout << p;
```

The first argument, `cout`, is an `ostream`

The second argument, `p`, is a `point`

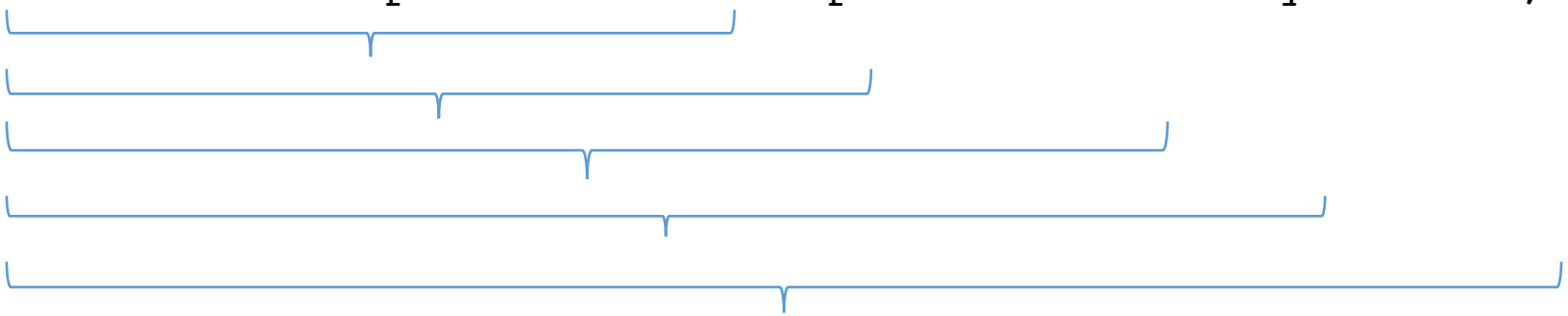
Overloading Output and Input Operators (Cont'd)

```
ostream& operator <<(ostream& outs, const point& source);
```

Also note that:

- The return type of the function is `ostream&`
- This return type means that the function returns an `ostream`
- Called a **reference return type**
- C++ permits the "**chaining**" of output statements such as the following:

```
cout << "The points are " << p << " and " << q << endl;
```



Overloading Output and Input Operators (Cont'd)

```
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.get_x( ) << " " << source.get_y( );
    return outs;
}


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;
}
```

Overloading Output and Input Operators (Cont'd)

- **In the input function, the input is sent to the private member variables! However, only member functions can access private member variables**

Two solutions:

- To write new member functions to set a point's coordinates and use these member functions within the input function's implementation
- You can grant special permission for the input function to access the private members of the point class – Called a **Friend Function**

Operator Overloading

Friend Functions

- A **friend function** is a function that **is not a member function**, but that still has access to the private members of the objects of a class
- To declare a friend function, the function's prototype is placed in a class definition, preceded by the keyword ***friend***

```
class point
{
public:
     ...
     // FRIEND FUNCTIONS
     friend istream& operator >>(istream& ins, point& target);

private:
     ...
};
```

The point class with a new friend

Friend Functions (Cont'd)

- **Even though the prototypes for friend functions appear in the class definition, friends are not member functions**
- Therefore, **it is not activated by a particular object of a class**
- **All of the information that the friend function manipulates must be present in its parameters**
- Friendship may be provided to any function, not just to operator functions
- Friendship should be limited to functions that are written by the programmer who implements the class

Operator Overloading

Friend Functions (Cont'd)

❑ Discuss about this example

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

class Box {
    double width;
public:
    friend void printWidth( Box box );
    void setWidth( double wid ) {width = wid;};
};

void printWidth( Box box ) {
    cout << "Width of box : " << box.width <<endl;
}

int main( ) {
    Box box;
    box.setWidth(10.0);

   printWidth( box );

    return 0;
}
```

The Standard Template Library (STL) and the Pair Class

The Standard Template Library and the Pair Class

- It is important for you to understand how to build and test your own data structures
- **However, you'll find that a suitable data structure has already been built for you to use in an application**

- In C++, **a variety of container classes,** called the **Standard Template Library (STL),** are available

The Pair Class

- Each pair object **can hold two pieces of data**

❑Example: integer and a double number; or a char and a bool value; or even a couple of throttles

```
#include <utility>    // Provides the pair class
using namespace std; // The pair is part of the std namespace

int main( )
{
    pair<int, double> p;
    p.first = 42;     // first is the member variable for the int piece
    p.second = 9.25; // second is the member variable for the double
    ...
```

The Pair Class (Cont'd)

```
template <class T1, class T2>
struct pair;
```

- This class couples together a pair of values, which may be of different types (T1 and T2)
- The individual values can be accessed through its **public members `first` and `second`**

- The **data types of these two pieces are specified in the angle brackets**, as part of the object's declaration
- We'll see more of these angle brackets (in future chapters) — called the **template instantiation**

The Pair Class (Cont'd)

```
template <class T1, class T2>
pair<T1,T2> make_pair (T1 x, T2 y);
```

- Constructs a pair object with its first element set to `x` and its second element set to `y`

The Standard Template Library and the Pair Class

The Pair Class (Cont'd)

Example

```
#include <utility>       // std::pair
#include <iostream>      // std::cout

int main () {
    std::pair <int,int> foo;
    std::pair <int,int> bar;

    foo = std::make_pair (1,2);
    bar = std::make_pair (0.5,'B'); //implicit conversion

    std::cout << "foo: " << foo.first << ", " << foo.second << '\n';
    std::cout << "bar: " << bar.first << ", " << bar.second << '\n';

    return 0;
}
```

```
Output:
foo: 1, 2
bar: 0, 66
```

Note: 66 is the decimal ASCII number of 'B'
`http://en.cppreference.com/w/cpp/language/ascii`

The Standard Template Library and the Pair Class

The Pair Class (Cont'd)

Example

```
#include <utility>       // std::pair, std::make_pair
#include <string>        // std::string
#include <iostream>      // std::cout

int main () {
    std::pair <std::string,int> university, myuniversity;

    university = std::make_pair("SCU",95053);

    myuniversity = university;

    std::cout << "My University: " << myuniversity.first << '\n';
    std::cout << "My University Zip: " << myuniversity.second << '\n';
    return 0;
}
```

```
Output:
My University: SCU
My University Zip: 95053
```

Summary

Summary

- Object-oriented programming (OOP) supports **information hiding** by placing data in packages called **objects**
- Objects are implemented via **classes**
- **Member functions** enable the manipulation of objects
- An **Abstract Data Type (ADT)/data structure/class:** A new data type, together with the functions to manipulate the type
  - The term **abstract** refers to the fact that we emphasize the abstract specification of **what has been provided**, **disassociated from any actual implementation**

Summary

- Information hiding is provided by **private** member variables
- Declaring a function as a **friend function** enables access to private members
- A **constructor** is a member function that is automatically called to initialize a data structure
- Using **namespace** avoids conflicts between different items with the same name

Summary

- **Header file** includes documentation and class definition
- **Implementation file** includes the implementation of the member functions
- C++ provides three common kinds of parameters: **value parameters**, **reference parameters**, **const reference parameters**
- C++ allows you to **overload operators** for your new data structures
- Many built-in classes are provided by the **Standard Template Library**

References

1) Data Structures and Other Objects Using C++, Michael Main, Walter Savitch, 4th Edition
2) Data Structures and Algorithm Analysis in C++, by Mark A. Weiss, 4TH Edition
3) C++: Classes and Data Structures, by Jeffrey Childs
4) http://en.cppreference.com
5) http://www.cplusplus.com
6) https://isocpp.org