Instance and Static Members

- **instance variable**: a member variable in a class. Each object has its own copy.

- **static variable**: one variable shared among all objects of a class

- **static member function**: can be used to access static member variable; can be called before any objects are defined

**Program 14.1**

```cpp
// This program demonstrates a static member variable.
#include <iostream>
#include <cassert>
using namespace std;

int main()
{
    // Define three Tree objects.
    Tree t1, t2, t3;
    // Display the number of Tree objects we have.
    cout << "We have " << t1.getObjectCount() << " trees in our program!"
    return 0;
}
```

**Program Output**

We have 3 trees in our program!

**Contents of Tree.h**

```cpp
// Tree class
class Tree
{
private:
    static int objectCount; // Static member variable.

public:
    // Constructor
    Tree()
    { objectCount++; }

    // Accessor function for objectCount
    int getObjectCount() const
    { return objectCount; }
};
```

**Contents of Tree.cpp**

```cpp
int Tree::objectCount = 0;
```

**Three Instances of the Tree Class, But Only One objectCount Variable**

- **objectCount variable**
  - (static)
  - `int Tree::objectCount = 0;`

- **Instances of the Tree class**
  - `oak`
  - `elk`
  - `pine`
static member function

- Declared with `static` before return type:
  
  ```cpp
  static int getObjectCount() const 
  { return objectCount; }
  ```

- Static member functions can only access static member data

- Can be called independent of objects:
  
  ```cpp
  int num = Tree::getObjectCount();
  ```

Friends of Classes

- **Friend**: a function or class that is not a member of a class, but has access to private members of the class

- A friend function can be a stand-alone function or a member function of another class

- It is declared a friend of a class with `friend` keyword in the function prototype

friend Function Declarations

- Stand-alone function:
  
  ```cpp
  friend void setAVal(intVal&, int);
  // declares setAVal function to be
  // a friend of this class
  ```

- Member function of another class:
  
  ```cpp
  friend void SomeClass::setNum(int num)
  // setNum function from SomeClass
  // class is a friend of this class
  ```

friend Class Declarations

- Class as a friend of a class:
  
  ```cpp
  class FriendClass
  {
    ...
  };
  ```

- Class as a friend of another class:
  
  ```cpp
  class NewClass
  {
    public:
    friend class FriendClass; // declares
    // entire class FriendClass as a friend
    // of this class
  };
  ```
14.3 Memberwise Assignment

- Can use `=` to assign one object to another, or to initialize an object with an object's data
- Copies member to member. *e.g.*,
  ```c++
  instance2 = instance1;
  ```
  means:
  copy all member values from `instance1` and assign to the corresponding member variables of `instance2`
- Use at initialization:
  ```c++
  Rectangle r2 = r1;
  ```

14.4 Copy Constructors

- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field
- Default copy constructor works fine in many cases
Copy Constructors

Problem: what if object contains a pointer?

```cpp
class SomeClass
{
    public:
        SomeClass(int val = 0)
        {value=new int; *value = val;}
        int getVal();
        void setVal(int);
    private:
        int *value;
};
```

What we get using memberwise copy with objects containing dynamic memory:

```cpp
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // also 13
```

Copy Constructor

Programmer-Defined Copy Constructor

- Allows us to solve problem with objects containing pointers:
  ```cpp
  SomeClass::SomeClass(const SomeClass &obj)
  {
      value = new int;
      *value = obj.value;
  }
  ```
- Copy constructor takes a reference parameter to an object of the class

Programmer-Defined Copy Constructor

- Each object now points to separate dynamic memory:
  ```cpp
  SomeClass object1(5);
  SomeClass object2 = object1;
  object2.setVal(13);
  cout << object1.getVal(); // still 5
  ```

Programmer-Defined Copy Constructor

- Since copy constructor has a reference to the object it is copying from,
  ```cpp
  SomeClass::SomeClass(const SomeClass &obj) it can modify that object.
  ```
- To prevent this from happening, make the object parameter `const`:
  ```cpp
  SomeClass::SomeClass
  (const SomeClass &obj)
  ```

```
```
14.5 Operator Overloading

Operator Overloading

- Operators such as -, +, and others can be redefined when used with objects of a class.
- The name of the function for the overloaded operator is operator followed by the operator symbol, e.g., operator+ to overload the + operator, and operator= to overload the = operator.
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it.
- Overloaded operator function definition goes with other member functions.

Prototype:

```
void operator=(const SomeClass &rval)
```

- Parameter for object on right side of operator: return type function name parameter for object on right side of operator.
- Operator is called via object on left side.

Invoking an Overloaded Operator

- Operator can be invoked as a member function:
  ```
  object1.operator=(object2);
  ```
- It can also be used in more conventional manner:
  ```
  object1 = object2;
  ```
Returning a Value

- Overloaded operator can return a value

```cpp
class Point2d
{
public:
    double operator-(const Point2d &right)
    { return sqrt(pow((x-right.x),2) + pow((y-right.y),2)); }
...
private:
    int x, y;
};
```

- Class Point2d
- Overload operator minus
- Class definition
- Private variables

```cpp
Point2d point1(2,2), point2(4,4);
// Compute and display distance between 2 points.
cout << point2 - point1 << endl; // displays 2.82843
```

The this Pointer

- `this`: predefined pointer available to a class's member functions
- Always points to the instance (object) of the class whose function is being called
- Is passed as a hidden argument to all non-static member functions
- Can be used to access members that may be hidden by parameters with same name

```cpp
class SomeClass
{
private:
    int num;
public:
    void setNum(int num)
    { this->num = num; }
    ...
};
```

Notes on Overloaded Operators

- Can change meaning of an operator
- Cannot change the number of operands of the operator
- Only certain operators can be overloaded
- Cannot overload the following operators:
  ```cpp
  ?: . .* :: sizeof
  ```

Overloading Types of Operators

- `++`, `--` operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a `bool` value
- Overloaded stream operators `>>, <<` must return reference to `istream`, `ostream` objects and take `istream`, `ostream` objects as parameters
Overloaded [] Operator

- Can create classes that behave like arrays, provide bounds-checking on subscripts
- Must consider constructor, destructor
- Overloaded [] returns a reference to object, not an object itself

Object Conversion

- Type of an object can be converted to another type
- Automatically done for built-in data types
- Must write an operator function to perform conversion
- To convert an FeetInches object to an int:
  FeetInches::operator int()
  (return feet);
- Assuming distance is a FeetInches object, allows statements like:
  int d = distance;

Aggregation

- Aggregation: a class is a member of a class
- Supports the modeling of ‘has a’ relationship between classes – enclosing class ‘has a’ enclosed class
- Same notation as for structures within structures

Aggregation

class StudentInfo
{
  private:
    string firstName, LastName;
    string address, city, state, zip;
    ...
};
class Student
{
  private:
    StudentInfo personalData;
    ...
};
See the Instructor, TextBook, and Course classes in Chapter 14.