14.1 Instance and Static Members

- **instance variable**: a member variable in a class. Each object (instance) 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;
  - cannot access instance variables

### Tree class declaration

```cpp
// Tree class
class Tree {
private:
    static int objectCount;
public:
    Tree();
    int getObjectCount();
};

// Definition of the static member variable, written outside the class.
int Tree::objectCount = 0;

// Member functions defined
Tree::Tree() {
    objectCount++;
}
int Tree::getObjectCount() {
    return objectCount;
}
```

### Program demo of static variable

```cpp
#include <iostream>
using namespace std;
#include "Tree.h"

int main() {
    Tree oak;
    Tree elm;
    Tree pine;
    cout << "We have " << pine.getObjectCount()
         << " Trees in our program.\n";
    return 0;
}
```

What will be the output?
Three Instances of the Tree Class, But Only One objectCount Variable

Instances of the Tree class

```cpp
string name1 = "Steve Jobs";
cout << "Name" << name1 << endl;
```

Static member function

- Declared with static before return type:
  ```cpp
  static int getObjectCount();
  ```

- Static member functions can access static member data only
  ```cpp
  int Tree::getObjectCount() {
    return objectCount;
  }
  ```

- Can be called independently of objects (use class name):
  ```cpp
  cout << "We have " << Tree::getObjectCount() << " Trees in our program.\n";
  ```

14.2 Friends of Classes

- **Friend**: a function or class that has access to private members of another class
  - even though it is NOT a member of that class

- The class must declare who its friends are in its class declaration

```
friend function example

```cpp
int AddToFriend(int x); //prototype
class HasFriend {
  friend int AddToFriend(int); //friend decl
private:
  int private_data;
public:
  HasFriend() {
    private_data = 5;
  }
};
int AddToFriend(int x) { //function def
  HasFriend var1;
  return var1.private_data + x; //error if not friend
}
int main() {
  cout << "Added 4 to Friend: " << AddToFriend(4) << endl;
}
```
friend class example

class NewFriend {
    public:
        int subtract(int);
};
class HasFriend {
    friend class NewFriend; //friend decl
    private:
        int private_data;
    public:
        HasFriend() { private_data = 5; }
};
int NewFriend::subtract(int x) {
    HasFriend var1;
    return var1.private_data - x; //error if not friend
}

int main() {
    NewFriend nf;
    cout << "Subtract 4 from Friend: " << nf.subtract(4) << endl;
}

14.3 Member-wise Assignment

- Can use = to
  - assign one object to another, or
  - initialize an object with another object’s data
- Copies member to member. e.g.,
  ```
  instance2 = instance1;
  ```
  means: copy all member values from instance1 and assign to the corresponding member variables of instance2
- Used at initialization: Time t2 = t1;

14.4 Copy Constructors

- Special constructor used when a newly created object is initialized using another object of same class.
  ```
  Time t1; Time t2 = t1; Time t3 (t1);
  ```
  Both of the last two use the copy constructor
- Also used when passing arguments by value
- The default copy constructor copies field-to-field (member-wise assignment).
- Default copy constructor works fine in many cases
IntCell declaration

- Problem: what if object contains a pointer?

```cpp
class IntCell
{
    private:
        int *storedValue;  // ptr to int
    public:
        IntCell (int initialValue);
        ~IntCell();
        int read () const;
        void write (int x);
};
```

IntCell Implementation

```cpp
#include "IntCell.h"
IntCell::IntCell (int initialValue) {
    storedValue = new int;
    *storedValue = initialValue;
}
IntCell::~IntCell() {
    delete storedValue;
}
int IntCell::read () const {
    return *storedValue;
}
void IntCell::write (int x) {
    *storedValue = x;
}
```

Problem with member-wise assignment

- What we get from member-wise assignment in objects containing dynamic memory (ptrs):

```cpp
IntCell object1(5);
IntCell object2 = object1;  // calls copy constructor
// object2.storedValue = object1.storedValue
object2.write(13);
cout << object1.read() << endl;
```

What is output? 5 13 or 13 13

Problem with member-wise assignment

- Why are they both changed to 13?
- Member-wise assignment does a shallow copy. It copies the pointer instead of allocating new memory and copying
- As a result, both objects point to the same location in memory
**Programmer-Defined Copy Constructor**

- Prototype and definition of copy constructor:
  ```cpp
  IntCell(const IntCell &obj) {  
    storedValue = new int;  
    *storedValue = obj.read(); // or *(obj.storedValue)
  }
  ```

- Copy constructor takes a reference parameter to an object of the class
  - otherwise it would use initialization to create the obj parameter, which would call the copy constructor for IntCell: this is an infinite loop

**Copy Constructor: limitations**

- Copy constructor is called ONLY during initialization of an object, NOT during assignment.
- If you use assignment with IntCell, you will still end up with member-wise assignment and a shared value:
  ```cpp
  IntCell object1(5);  
  IntCell object2(0);  
  object2 = object1;  
  object2.write(13);  
  cout << object1.read() << endl;  
  cout << object2.read() << endl;  
  ```

**14.5 Operator Overloading**

- Operators such as =, +, and others can be redefined to work over objects of a class
- The name of the function defining the overloaded operator is `operator` followed by the operator symbol:
  - `operator+` to overload the + operator, and `operator=` to overload the = operator
- Just like a regular member function:
  - Prototype goes in the class declaration
  - Function definition goes in implementation file
Operator Overloading

- Prototype in class declaration:
  ```cpp
global operator= (const IntCell &rhs);
```
- `operator=` is the function name
- `IntCell &rhs` is the parameter for the right hand side of operator.

- The operator function is defined from the perspective of the object on the left side of the =
- The operator function is called via object on left side

Invoking an Overloaded Operator

- Operator can be invoked as a member function:
  ```cpp
  object1.operator=(object2);
  ```
- It can also be invoked using the more conventional syntax:
  ```cpp
  object1 = object2;
  ```
- Both call the same `operator=` function, from the perspective of object1

Overload = for IntCell

```cpp
class IntCell {
  private:
    int *value;
  public:
    IntCell(const IntCell &obj);
    IntCell(int);
    ~IntCell();
    int read() const;
    void write(int);
    void operator= (const IntCell &rhs);
};
```

```cpp
void IntCell::operator= (const IntCell &rhs) {
    write(rhs.read());
}
```

//in a driver:
```cpp
IntCell object1(5), object2(0);
object2 = object1;
object2.write(13);
cout << object1.read() << endl;
```

Output: 5

Returning a Value

- An overloaded operator can return a value

```cpp
int Time::operator- (Time &right) {  //Note: 12%12 = 0
  return (hour%12)*60 + minute -
    ((right.hour%12)*60 + right.minute);
}
```

//in a driver:
```cpp
Time time1(12,20), time2(4,40);
int minutesDiff = time2 - time1;
cout << minutesDiff << endl;
```

Output: 260
Overloaded Operators

• Cannot change the number of operands of the operator or the return type

• Overloaded relational operators should return a bool value

• I recommend avoiding using overloaded operators in expressions with << (assign the result of the operation to a variable, then output the variable).

```cpp
class Time
{
    private:
    int hour;
    int minute;
    void addHour();

    public:
    Time();
    Time(int);
    Time(int, int);
    void addMinute(); // adds one minute
    void addMinute(int); // adds n minutes
    int getHour();
    int getMinute();

    int operator-(Time &right);
    bool operator== (const Time &right);
    bool operator< (const Time &right);

    void setHour(int);
    void setMinute(int);
    string display();
};
```

```cpp
bool Time::operator== (const Time &right) {
    if (hour == right.hour &&
        minute == right.minute)
        return true;
    else
        return false;
}

bool Time::operator< (const Time &right) {
    if (hour == right.hour)
        return (minute < right.minute);
    return (hour%12) < (right.hour%12);
}

// in a driver:
Time time1(12, 20), time2(12, 21);
if (time1<time2) cout << "correct" << endl;
time1.addMinute();
if (time1==time2) cout << "correct again" << endl;
```