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;
  - normal functions can access static member variables, too
  - but it cannot access instance variables
  - can be called like a standalone function

### 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";
}
```

What will be the output?
Three instances of the Tree class, but only one objectCount variable

Instances of the Tree class

```cpp
int Tree::getObjectCount() {
    return objectCount;
}
```

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";
```

14.3 Member-wise Assignment

- Can use = to
  - assign (copy) one object to another, or
  - initialize an object with another object's data
- Copies member to member. e.g.,

```cpp
instance2 = instance1;
```

means: copy all member values from instance1 and assign to the corresponding member variables of instance2

- Also used at initialization:

```cpp
Time t2 = t1;
```

Member-wise assignment: demo

```cpp
Time t1(10, 20);
Time t2(12, 40);
cout << "t1: " << t1.display() << endl;
cout << "t2: " << t2.display() << endl;
t2 = t1;
cout << "t1: " << t1.display() << endl;
cout << "t2: " << t2.display() << endl;
```

Output:
```
t2 = t1; //equivalent to:
t2.hour = t1.hour;
t2.minute = t1.minute;
```
14.4 Copy Constructors

- Special constructor used when a newly created object is initialized using another object of the same class.
- Used implicitly when passing arguments by value
- The default copy constructor copies field-to-field (member-wise assignment).
- Default copy constructor works fine in many cases

Problem: what if the 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 object1(5);
IntCell object2 = object1; // calls copy constructor
// object2.storedValue = object1.storedValue
object2.write(13);
cout << object1.read() << endl;
cout << object2.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’s address instead of allocating new memory and copying the value.
- As a result, both objects point to the same location in memory

![Diagram showing object1 and object2 pointing to the same location in memory](image)

Member-wise assignment does a shallow copy. It copies the pointer’s address instead of allocating new memory and copying the value. As a result, both objects point to the same location in memory.

![Diagram showing object1 and object2 pointing to the same location in memory](image)

Programmer-Defined Copy Constructor

- Prototype and definition of copy constructor:

```cpp
IntCell(const IntCell &obj);  // Add to class declaration

IntCell::IntCell(const IntCell &obj) {
    storedValue = new int;
    *storedValue = obj.read();  // or *(obj.storedValue)
    // or even: write(obj.read);
}
```

- Copy constructor takes a reference parameter to an object of the class
- Otherwise, pass-by-value would use the copy constructor to initialize the obj parameter, which would call the copy constructor: this is an infinite loop

![Diagram showing object1 and object2](image)

Each object now points to separate dynamic memory:

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

Output:

```
5
13
```

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:

```
IntCell object1(5);
IntCell object2(0);
object2 = object1;  // object2.storedValue = object1.storedValue
object2.write(13);
cout << object1.read() << endl;
cout << object2.read() << endl;
```

Output:

```
13
13
```
14.5 Operator Overloading

- Operators such as =, +, <, ..., can be defined to work for objects of a programmer-defined class
- The name of the function defining the over-loaded operator is `operator` followed by the operator symbol:
  - `operator+` to define the + operator, and
  - `operator=` to define the = operator
- Just like a regular member function:
  - Prototype goes in the class declaration
  - Function definition goes in implementation file

Calling an Overloaded Operator

- The operator function is called via the object on the left side
- It can be called like a normal member function:
  ```cpp
  int minutes = object1.operator-(object2);
  ```
- It can also be called using the more conventional operator syntax:
  ```cpp
  int minutes = object1 - object2;
  ```
- Both call the same `operator-` function, from the perspective of `object1`

Example: minus for Time objects

```cpp
class Time {
    private:
        int hour, minute;
    public:
        int operator- (Time right); //Note: 12%12 = 0
    }

int Time::operator- (Time right) {
    return (hour%12)*60 + minute -
            ((right.hour%12)*60 + right.minute);
}

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

Output: 260
Overloading == and < for Time

```cpp
class Time
{
  private:
    int hour;
    int minute;
  public:
    Time();
    Time(int);
    Time(int, int);
    void addHour();
    void addMinute(); // adds one minute
    void addMinute(int); // adds n minutes
    void getHour();
    void getMinute();

    int operator- (Time right);
    bool operator== (Time right); // compares Time objects
    bool operator< (Time right); // compares Time objects

    void setHour(int);
    void setMinute(int);

    string display();
};
```

Overloading + for Time

```cpp
class Time
{
  private:
    int hour;
    int minute;
  public:
    Time operator+ (Time right); // Note: 12%12 = 0
    void addHour();
    void addMinute(); // adds one minute
    void addMinute(int); // adds n minutes
    void getHour();
    void getMinute();

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

    void setHour(int);
    void setMinute(int);

    string display();
};
```

Overload = for IntCell

```cpp
class IntCell
{
  private:
    int *storedValue;
  public:
    IntCell(const IntCell &obj);
    IntCell(int);
    ~IntCell();
    int read() const;
    void write(int);
    IntCell& operator= (IntCell rhs); // Copy assignment

    void IntCell::operator= (IntCell rhs) {
        write(rhs.read());
    }

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