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

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;`

static member function

- Declared with static before return type:
  ```
  static int getObjectCount();
  ```
- Static member functions can access static member data only
  ```
  int Tree::getObjectCount() {
      return objectCount;
  }
  ```
- Can be called independently of objects (use class name):
  ```
  cout << "We have " << Tree::getObjectCount() << "Trees in our program.\n";
  ```

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;
t1: 10:20
t2: 10:20
```
14.4 Copy Constructors

- Special constructor used when a newly created object is initialized using another object of the same class.

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

```
Time t1;
Time t2 = t1;
Time t3 (t1);
```

IntCell declaration

- Problem: what if object contains a pointer?

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

IntCell Implementation

```
#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):

```
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 instead of allocating new memory and copying
- As a result, both objects point to the same location in memory

![Diagram showing object1 and object2 with storedValue pointing to the same location]

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); // Add to class declaration
  ```

  ```cpp
  IntCell::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 the copy constructor to initialize the obj parameter, which would call the copy constructor: this is an infinite loop

![Diagram showing object1 and object2 with storedValue pointing to separate locations]

Each object now points to separate dynamic memory:

```cpp
IntCell object1(5);
IntCell object2 = object1; // now calls MY copy constr
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:

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

Output: 13 13
14.5 Operator Overloading

- Operators such as =, +, <, and others can be defined to work for objects of a user-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

Invoking an Overloaded Operator

- Operator can be invoked (called) as a member function:
  ```
  int minutes = object1.operator-(object2);
  ```
- It can also be invoked using the more conventional syntax:
  ```
  int minutes = object1 - object2;
  ```
- Both call the same `operator-` function, from the perspective of `object1`
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== (Time right);
        bool operator< (Time right);
    void setHour(int);
    void setMinute(int);
    string display();
};

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

bool Time::operator< (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;

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

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;  // Output: 5

Now = for IntCell will not use member-wise assignment