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 before any objects are defined

```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

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

- 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: `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:
```
t1: 10:20
```
```cpp
t2 = t1; //equivalent to:
t2.hour = t1.hour;
t2.minute = t1.minute;
```

Output:
```
t1: 10:20
t2: 12:40
t1: 10:20
```
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

IntCell declaration

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 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;
cout << object2.read() << endl;
```

What is output? 5 or 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
- As a result, both objects point to the same location in memory

Programmer-Defined Copy Constructor

- Prototype and definition of copy constructor:
  ```c++
  IntCell(const IntCell &obj);
  ```
  Add to class declaration
  ```c++
  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, pass-by-value would use the copy constructor to initialize the obj parameter, which would call the copy constructor: 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:
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.

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

```
int operator- (Time right) {    //Note: 12%12 = 0
    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

Example: minus for Time objects
Overloading == and < for Time

```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== (Time right); //adds n minutes
    bool operator< (Time right); //adds n minutes
    void setHour(int);
    void setMinute(int);
    string display();
};
```

Overloading == and < for Time

```cpp
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;
```

Overloading + for Time

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

Time Time::operator+ (Time right) { //Note: 12%12 = 0
    int totalMin = (hour % 12) * 60 + minute +
                   (right.hour % 12) * 60 + right.minute;
    int h = totalMin / 60; //integer division, total hours
    h = h % 12; //keep it between 0 and 11
    if (h == 0) h = 12; //convert 0:xx to 12:xx
    int totalMin % 60); //create new time obj
    return result;
} //in a driver:
Time t1(12, 5);
Time t2(2, 50);
Time t3 = t1 + t2;
cout << t3.display() << endl;
```

Output: 2:55

Overloading + for Time

```cpp
class IntCell {
private:
    int *value;
public:
    IntCell(const IntCell &obj);
    IntCell(int);
    ~IntCell();
    int read() const;
    void write(int);
    void operator= (IntCell rhs); //Now = for IntCell will not use member-wise 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;
```

Output: 5