### Overloading the `<<` Operator

In order to overload the `<<` operator for printing Clock objects, we add the following code to the Clock class:

```cpp
class Clock
{
public:
    ...
    friend ostream &operator<<(ostream &out, Clock C);
    ...);

    ostream &operator<<(ostream &out, Clock C)
    {
        return (out << C.mins << " : " << C.secs);
    }
};
```

With this addition, we can output Clock values as follows:

```cpp
int main()
{
    Clock C1(59), C2(600);
    cout << "Initial times:
" << C1 << "
" << C2;
    C1++;
    C2++;
    cout << "After one second times are:
" << C1 << "
" << C2;
    Clock C3 = C1 + C2;
    cout << "The sum of these times is:
" << C3;
    Clock C4 = 2*C3;
    cout << "Multiplied by two is:
" << C4;
}
```

The output looks like before:

```
Initial times:
0:59
10:0
After one second times are:
1:0
10:1
The sum of these times is:
11:1
Multiplied by two is:
22:2
```
Overloading the << Operator

```cpp
class Matrix
{
public:
    ...
   ostream &operator<<(ostream &out, const Matrix &m)
    {
        out << endl;
        for (int y = 0; y < m.dy; y++)
        {
            for (int x = 0; x < m.dx; x++)
                out << m.p[x][y] << "\t";
            out << endl;
        }
        return out;
    }
};
```

Now the following main function...

```cpp
int main()
{
    Matrix myMatrix(3, 3);
    myMatrix.Element(2, 2) = 1;
    cout << myMatrix << endl;
}
```

... prints the following output:

```
0 0 0
0 0 0
0 0 1
```

Overloading the () Operator

You may remember that we defined the following member function to access individual elements of a matrix:

```cpp
long &Matrix::Element(int x, int y)
{
    assert(x >= 0 && x < dx && y >= 0 && y < dy);
    return p[x][y];
}
```

Whenever we want to manipulate an element, we have to explicitly call this function, for example:

```cpp
Matrix myMatrix(3, 3);
myMatrix.Element(1, 0) = 3;
```

It is much more convenient to overload the function call operator for the Matrix class. This will allow us to access Matrix elements by just applying the () operator to a Matrix object.

We have to add the following code:

```cpp
class Matrix
{
public:
    ...
    long &operator()(int x, int y); // return reference to an element
    ...
};
```

(continued on next slide)

Now the following main function...

```cpp
int main()
{
    Matrix myMatrix(3, 3);
    myMatrix(2, 2) = 1;
    cout << myMatrix << endl;
}
```

... prints the following output:

```
0 0 0
0 0 0
0 0 1
```

Overloading the = Operator

Given the current state of our Matrix class, is the following main() function syntactically correct?

```cpp
int main()
{
    Matrix myMatrix(2, 2), yourMatrix(2, 2), theirMatrix(4, 4);
    myMatrix(1, 1) = 1;
    yourMatrix = myMatrix;
    theirMatrix = myMatrix;
    myMatrix(0, 0) = 1;
    cout << myMatrix << yourMatrix << theirMatrix;
    return 0;
}
```

Yes, it is. The compiler provides a default assignment operator that performs copying of all members’ values. However, it acts in a counterintuitive way and also creates memory leaks.
Overloading the = Operator

What does the output of the program look like?

1 0
0 1
1 0
0 1
1 0
0 1

The pointer **p in yourMatrix and theirMatrix is simply copied from myMatrix, that is, it points to the same memory location. Therefore, the three matrices are identical; whenever we change the values of elements in one matrix, we automatically change them in the other two matrices as well. The three matrices are not independent copies.

It is important to note that, besides being counterintuitive, this behavior also leads to memory leaks. When the following line of code is being executed...

Matrix yourMatrix(2, 2);

... memory is being allocated on the heap, and yourMatrix.p points to its location. When we then do the following assignment...

yourMatrix = myMatrix;

... it implies that yourMatrix.p is set to myMatrix.p, and thus no pointer can access the memory reserved for yourMatrix anymore. The memory remains allocated on the heap but cannot be used or freed anymore, giving us a memory leak.

If this occurs in a function that is repeatedly called, the reserved memory will accumulate and eventually interfere with program execution.

To avoid this, we have to provide our own assignment operator that also copies the actual elements of a matrix and lets the pointer point to the copied data. We add the following code:

```cpp
class Matrix
{
public:
    ...
    Matrix &operator=(const Matrix &m);
    ...
};
```

(continued on next slide)

Overloading the = Operator

Matrix &Matrix::operator=(const Matrix &m)
{
    if (this != &m)
    {
        assert(dx == m.dx && dy == m.dy);
        for (int i = 0; i < dx; i++)
            for (int j = 0; j < dy; j++)
                p[i][j] = m.p[i][j];
    }
    return *this;
}

Notice: We only want to allow copying matrices of identical size. Also, if a matrix is to be copied to itself, we do not need to do anything.

Given the new state of our Matrix class with the overloaded = operator, what is the output of the following function?

```cpp
int main()
{
    Matrix myMatrix(2, 2), yourMatrix(2, 2);
    myMatrix(1, 1) = 1;
    yourMatrix = myMatrix;
    myMatrix(0, 0) = 1;
    cout << myMatrix << yourMatrix;
}
```

It is: 1 0
0 1
0 0
0 1

The Copy Constructor

By the way, a similar problem happens when we pass a class object as a call-by-value argument. Example:

```cpp
void OurFunction(int i, Matrix m);
```

Similar to the default assignment operator, the compiler will provide a default copy constructor. This copy constructor will create a local copy m of the matrix that we pass to the function, and will copy the values of all class members to the members of m. Again, the pointer variables will be copied, but not the memory locations that they point at.
The Copy Constructor

Even worse, once OurFunction(int i, Matrix m) has finished its execution, the local copy Matrix m will automatically be deleted by calling its destructor. However, since m.p points to the location where the data of the matrix in the function call (the matrix external to the function) are stored, the data of that matrix will also be deleted! In other words, calling OurFunction will destroy the matrix that we use as an argument!

To solve these problems, we have to provide our own copy constructor.

We add the following code:

class Matrix
{
public:
    Matrix(const Matrix &m);
    // (continued on next slide)
};

The Copy Constructor

Matrix::Matrix(const Matrix &m) : dx(m.dx), dy(m.dy)
{
    p = new long*[dx]; // create array of pointers to long integers
    for (int i = 0; i < dx; i++)
        p[i] = new long[dy]; // for each pointer, create array of l.i.s
    for (int j = 0; j < dy; j++)
        p[i][j] = m.p[i][j];
}

References as Function Arguments

C++:

```cpp
void increaseMe(int &x)
{
    x++;
}
```

```cpp
int main()
{
    int y = 3;
    increaseMe(y);
    cout << y;
    return 0;
}
```

Output: 4

C:

```c
void increaseMe(int *x)
{
    (*x)++;
}
```

```c
int main()
{
    int y = 3;
    increaseMe(&y);
    printf("%d", y);
    return 0;
}
```

Output: 4

Class Templates

- C++ uses the keyword `template` to provide parametric polymorphism.
- This allows the same code to be used with respect to various types, where the type is a parameter of the code body.
- This is a form of generic programming.
- It allows us to reuse code in a simple and type-safe manner.
- We will use this technique to define a template for the Matrix class so that we can instantiate matrices for different types of matrix elements.

```cpp
template<class TYPE>
class Matrix
{
public:
    Matrix(int sizeX, int sizeY, TYPE initValue = TYPE());
    Matrix(const Matrix &m);
    ~Matrix();
    int GetSizeX() const { return dx; }
    int GetSizeY() const { return dy; }
    TYPE &operator()(int x, int y); // return reference to an element
    Matrix &operator=(const Matrix &m);
    template<class T>  // for friends, separate template declaration necessary
    friend ostream &operator<<(ostream &out, const Matrix<T> &m);
private:
    TYPE **p; // pointer to a pointer to a TYPE
    int dx, dy;
};
```
template<class TYPE>
Matrix<TYPE>::Matrix(int sizeX, int sizeY, TYPE initValue) : dx(sizeX),
dy(sizeY)
{
    assert(sizeX > 0 && sizeY > 0);
p = new TYPE*[dx]; // create array of pointers to TYPE
assert(p != 0);
for (int i = 0; i < dx; i++)
{
    p[i] = new TYPE[dy]; // for each pointer, create array of TYPE
    assert(p[i] != 0);
    for (int j = 0; j < dy; j++)
        p[i][j] = initValue;
}
}

template<class TYPE>
Matrix<TYPE>::Matrix(const Matrix<TYPE> &m) : dx(m.dx), dy(m.dy)
{
p = new TYPE*[dx]; // create array of pointers to TYPE
assert(p != 0);
for (int i = 0; i < dx; i++)
{
p[i] = new TYPE[dy]; // for each pointer, create array of TYPE
    assert(p[i] != 0);
    for (int j = 0; j < dy; j++)
        p[i][j] = m.p[i][j];
}
}

template<class TYPE>
Matrix<TYPE>::~Matrix()
{
    for (int i = 0; i < dx; i++)
        delete [] p[i]; // delete arrays of TYPE
    delete [] p; // delete array of pointers to TYPE
}

template<class TYPE>
TYPE &Matrix<TYPE>::operator()(int x, int y)
{
    assert(x >= 0 && x < dx && y >= 0 && y < dy);
    return p[x][y];
}

int main()
{
    Matrix<char> myMatrix(2, 2, 'a');
    myMatrix(0, 0) = 'b';
    cout << myMatrix << "\n";
    return 0;
}
Output:
b a
a a
int main()
{
    Matrix<Clock> clockMatrix(4, 4, 620);
    cout << clockMatrix << "\n\n";
    ++clockMatrix(1, 1);
    clockMatrix(3, 3) = 2 * clockMatrix(2, 2);
    cout << clockMatrix << "\n\n";
}

Output:

```
10:20 10:20 10:20 10:20
10:20 10:20 10:20 10:20
10:20 10:20 10:20 10:20
10:20 10:20 10:20 10:20
10:20 10:20 10:20 10:20
10:20 10:20 10:20 10:20
10:20 10:20 10:20 10:20
10:20 10:20 10:20 20:40
```

(The code for this example can be found as clock_matrix.cpp on the course homepage.)