A Singly Linked List

We want to be able to perform the following list operations:

- **Prepend**: adds an element to the front of the list
- **First**: returns the first element in the list
- **Print**: prints the list contents
- **Del**: deletes the first element in the list
- **Release**: destroys the list

```cpp
struct SListElem
{
    char data;
    SListElem *next;
};
class SList
{
public:
    SList(): h(0) {} // 0 denotes empty list
    ~SList() { Release(); }
    void Prepend(char c);
    void Del();
    SListElem *First() const { return h; }
    void Print() const;
    void Release();
private:
    SListElem *h; // head of SList
};
```

```cpp
void SList::Prepend(char c)
{
    SListElem *temp = new SListElem; // create new element
    assert( temp != 0);
    temp->next = h; // link to SList
    temp->data = c;
    h = temp; // update head of SList
}
```

```cpp
void SList::Del()
{
    SListElem *temp = h;
    if (temp != 0)
    {
        h = h->next; // let head point to 2nd element
        delete temp; // delete 1st element
    }
}
```

```cpp
void SList::Print() const
{
    SListElem *temp = h;
    while (temp != 0) // detect end of Slist
    {
        cout << temp->data << " -> ";
        temp = temp->next; // temp proceeds to next element
    }
    cout << "NULL\n";
}
```

```cpp
void SList::Release()
{
    while (h != 0) // as long as there are elements...
    {
        Del(); // ... delete them.
    }
}
```
A Singly Linked List

```cpp
int main()
{
    SList MyList;
    MyList.Prepend('S');
    MyList.Prepend('T');
    MyList.Prepend('A');
    MyList.Prepend('R');
    MyList.Prepend('T');
    MyList.Del();
    MyList.Print();
}
```

Output: R -> A -> T -> S -> NULL

---

A Two-Dimensional Array

- We are now going to implement a dynamic and safe two-dimensional array.
- We will use a pointer-to-pointer-to-base type structure for this implementation.

```cpp
class Matrix
{
public:
    Matrix(int sizeX, int sizeY);
    ~Matrix();
    int GetSizeX() const { return dx; }
    int GetSizeY() const { return dy; }
    long &Element(int x, int y); // return reference to an element
    void Print() const;
private:
    long **p; // pointer to a pointer to a long integer
    int dx, dy;
};
```

```cpp
Matrix::Matrix(int sizeX, int sizeY) : dx(sizeX), dy(sizeY)
{
    assert(sizeX > 0 && sizeY > 0);
    p = new long*[dx]; // create array of pointers to long integers
    assert(p != 0);
    for (int i = 0; i < dx; i++)
        p[i] = new long[dy];  // for each pointer, create array of long integers
    assert(p[i] != 0);
    for (int j = 0; j < dy; j++)
        p[i][j] = 0;
}
```

```cpp
Matrix::~Matrix()
{
    for (int i = 0; i < dx; i++)
        delete [] p[i]; // delete arrays of long integers
    delete [] p; // delete array of pointers to long integers
}
```

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

```cpp
void Matrix::Print() const
{
    cout << endl;
    for (int y = 0; y < dy; y++)
        for (int x = 0; x < dx; x++)
            cout << p[x][y] << " ";
    cout << endl;
}
```
A Two-Dimensional Array

```cpp
int main()
{
    Matrix MyMatrix(3, 5);
    MyMatrix.Element(1, 1) = 88;
    MyMatrix.Element(2, 2) = 11;
    MyMatrix.Element(2, 4) = MyMatrix.Element(1, 1) + MyMatrix.Element(2, 2);
    cout << MyMatrix.Element(2, 4) << endl;
    MyMatrix.Print();
}
```

Output:
```
99
0 0 0
0 88 0
0 0 11
0 0 0
0 0 99
```

ADT Conversions

Previously we talked about constructors as conversions. For example, in our class ModInt we provided the following constructor:

```cpp
ModInt(int i = 0): v(i % modulus) {}
```

It is used automatically to convert values of type int into values of type ModInt:

```cpp
int i = 39;
ModInt m;
m = i;     // illegal operation
```

However, this constructor does not tell the compiler how to convert a ModInt value into an int value:

```cpp
int i = 39;
ModInt m;
i = m;     // illegal operation
```

It is not possible for the user to add a constructor to a built-in type such as int or double. How can we solve this problem?

ADT Conversions

The solution is to provide a special conversion function inside the user-defined class. The general form of such a member function is:

```cpp
operator type() { … };
```

Such a member function must
- be nonstatic,
- have no parameters,
- have no declared return type,
- return an expression of the designated type.

For example, we could add such a special conversion function to the ModInt class:

```cpp
ModInt::operator int()
{
    return v;
}
```

Then the following code is correct:

```cpp
int i = 39;
ModInt m;
i = m;
```

Here, i receives the value 0.
Overloading Operators

We now know how to use the `operator` keyword to define a type-conversion member function. We will now take a look at how to use the `operator` keyword to overload the built-in C++ operators.

We have already seen that overloading can give a function name a variety of meanings, depending on its arguments.

Overloading operators

• can give additional meanings to the built-in operators such as `+`,
• allows infix expressions of both ADTs and built-in types,
• leads to shorter, more readable programs.

Example:

```cpp
class Foo
{
public:
    Foo operator-();  // unary minus: -Foo
    Foo operator-(int);   // binary minus: Foo – int
    Foo operator-(Foo);  // binary minus: Foo – Foo
};
```

```cpp
Foo operator-(int, Foo); // binary minus: int - Foo
```

Unary Operator Overloading

Example:

```cpp
class Clock
{  
    public:
        Clock(unsigned long i);
    void Print() const { cout << mins << ":" << secs << endl; }
    void Tick(); // add one second
    Clock operator++() { Tick(); return *this; }

    private:
        unsigned long totSecs, secs, mins;
};
```

```cpp
Clock::Clock(unsigned long i)
{
    totSecs = i;
    secs = totSecs % 60; // convert into minutes-seconds format
    mins = (totSecs / 60) % 60;
}
```

```cpp
void Clock::Tick()
{
    Clock Temp = Clock(++totSecs);
    secs = Temp.secs;
    mins = Temp.mins;
}
```

```cpp
int main()
{
    Clock C1(59), C2(600);
    cout << "Initial times:" << endl;
    C1.Print();
    C2.Print();
    cout << endl;
    ++C1; // increase times by one second
    ++C2;
    cout << "After one second times are:" << endl;
    C1.Print();
    C2.Print();
    cout << endl;
}
```

 Unary Operator Overloading

Output:

Initial times:
0:59
10:0

After one second times are:
1:0
10:1
Unary Operator Overloading

We could also have overloaded the prefix `++` by using an **ordinary** function:

```cpp
Clock operator++(Clock& C) {
    C.Tick();
    return C;
}
```

Binary Operator Overloading

We continue with our clock example and show how to overload **binary** operators. When a binary operator is overloaded using a member function, it has
- as its first argument the implicitly passed class variable,
- as its second argument the single argument-list parameter.

**Friend functions** and **ordinary functions** have both arguments specified in the parameter list. Of course, ordinary functions cannot access private members.

```
Binary Operator Overloading

class Clock {
    ...
    friend Clock operator+(Clock C1, Clock C2);
};

Clock operator+(Clock C1, Clock C2) {
    return (C1.totSecs + C2.totSecs);
}

Here, the **Clock constructor** provides the implicit conversion from unsigned long to Clock.
```

```
Binary Operator Overloading

Analogously, we can overload the multiplication operator:

```
class Clock {
    ...
    friend Clock operator*(unsigned long factor, Clock C);
};

Clock operator*(unsigned long factor, Clock C) {
    return (factor*C.totSecs);
}
```

Notice that this function demands a fixed ordering (unsigned long * Clock).

```
Binary Operator Overloading

To avoid this, we can add a second overloaded function:

```
Clock operator*(Clock C, unsigned long factor) {
    return (factor*C);
}
```

Notice that this second function overloading `*` is defined in terms of the first one. Therefore, we do not need to make it a friend function.

```
Binary Operator Overloading

Testing the new functions:

```cpp
int main() {
    ...
    cout << "The sum of these times is:" << endl;
    Clock C3 = C1 + C2;
    C3.Print();
    cout<< endl;
    cout << "Multiplied by two is:" << endl;
    Clock C4 = 2*C3;
    C4.Print();
    cout<< endl;
}
```
Binary Operator Overloading

Output:

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

Do you remember the Clock example?

class Clock
{
    public:
        Clock(unsigned long i = 0);
        void Print() const { cout << mins << ":" << secs << endl; }
        void Tick(); // add one second
    Clock operator++() { Tick(); return *this; }
    private:
        unsigned long totSecs, secs, mins;
};

Overloading the << Operator

Printing a time value works as follows:

```
int main()
{
    Clock C1(59), C2(600);
    cout << "Initial times: " << endl;
    C1.Print();
    C2.Print();
}
```

It would be more convenient to use the << operator for creating output, so we are going to overload the << operator for Clock objects.

Overloading the << Operator

How does the << operator work?

Example:

```
int main()
{
    string text1("John is "), text2("years old.");
    int age = 43;
    cout << text1 << age << text2;
}
```

...