ADT Conversions

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

\[
\text{ModInt}(\text{int } i = 0): v(i \mod \text{modulus})
\]

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

\[
\text{int } i = 39;
\text{ModInt } m;
\text{m} = i; \quad \text{// legal operation}
\]

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

\[
\text{int } i = 39;
\text{ModInt } m;
i = m; \quad \text{// illegal operation}
\]

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?

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

\[
\text{operator type}() \{ \ldots \}
\]

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:

\[
\text{ModInt: operator int()}
\{
\text{return v;}
\}
\]

Then the following code is correct:

\[
\text{int } i = 39;
\text{ModInt } m;
i = m;\quad \text{// illegal operation}
\]

Here, \( i \) receives the value 0.

Overloading Operators

We now know how to use the \texttt{operator} keyword to define a type-conversion member function.

We will now take a look at how to use the \texttt{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.

\textbf{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.

\[
\text{class Foo}
\{
\text{public:}
\text{Foo operator-(int, Foo);} \quad \text{// binary minus: int - Foo}
\}
\]
Unary Operator Overloading

Example:

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;
};

Unary Operator Overloading

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

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

Unary Operator Overloading

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();
}

Unary Operator Overloading

Output:

Initial times:
0:59
10:0

After one second times are:
1:0
10:1

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.

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).

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.

Testing the new functions:

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

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

Printing a time value works as follows:

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

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:

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

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);
    ...
};
```

```cpp
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:\n" << C1 << "\n" << C2;
    C1++;  
    C2++;  
    cout << "\nAfter one second times are:\n" << C1 << "\n" << C2;
    Clock C3 = C1 + C2;
    cout << "\nThe sum of these times is:\n" << C3;
    Clock C4 = 2*C3;
    cout << "\nMultiplied by two is:\n" << C4 << "\n";
}
```

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

Overloading the `<<`, `()`, and `=` operators can be nicely demonstrated with the `Matrix` example.

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

```cpp
class Matrix {
    public:
        ...
        friend ostream &operator<<(ostream &out, const Matrix &m);
        ...
};
(continued on next slide)
```

```
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;
}
```