Objects Acting Like Values

Laurent Mathy

Object-Oriented Programming Projects

April 20, 2020

Doing more with our objects

So far we can *create*, *destroy*, *copy* and *assign* our objects.

Most built-in types support more operations:

- Rich set of **operators** (`+`, `*`, `-=`, *etc.*)
- **Conversions** between types (*e.g.* `int` to `double`)

Simple string class

```
class Str {
public:
    typedef std::vector<char>::size_type size_type;

    Str() { }

    Str(size_type n, char c): data(n, c) { }

    Str(const char* cp) {
        std::copy(cp, cp + std::strlen(cp),
                  std::back_inserter(data));
    }

    template<class In> Str(In i, In j) {
        std::copy(i, j, std::back_inserter(data));
    }

private:
    std::vector<char> data;
};
```

`Str` constructors

- Default constructor implicitly invokes default `vector` constructor:
  - Must be explicitly defined. (Why?)
  - Does exactly what the synthesised default would do.
- Constructor that takes a size and a character uses corresponding `vector` constructor as initialiser.
- Two other constructors do not have initialisers:
  - `data` is first implicitly initialised as empty `vector`.
  - Both then append to `data`.
  - Last constructor is a **`template`** so it defines a **family** of constructors. *E.g.* it can construct a `Str` from an array of **`char`**, from a `vector<`**`char`**`>`, from a substring, *etc.*

Copy, assignment and destructor

Our class does not define any copy constructor, assignment operator or destructor.

Why?

Copy, assignment and destructor

Our class does not define any copy constructor, assignment operator or destructor.

Why?

Because the defaults work!

The `Str` class itself does no memory management.

The synthesised operations will call corresponding `vector` operations: that's exactly what we want!

Bottom line: `Str` does not need a destructor – it would have nothing to do.

If you don't need a destructor, you don't need explicit copy constructor or assignment operator (rule of three).

Automatic conversions

```
Str s = "hello"; // Initialise s
```

Requires the copy constructor that takes a `const Str&` as argument.

```
s = "bye"; // Assign a new value to s
```

Requires the assignment operator that takes a `const Str&` as argument.

Automatic conversions

```
Str s = "hello"; // Initialise s
```

Requires the copy constructor that takes a `const Str&` as argument.

```
s = "bye"; // Assign a new value to s
```

Requires the assignment operator that takes a `const Str&` as argument.

But both statements specify a `const char*`, not the expected `const Str&`!

Automatic conversions (2)

We already have a constructor that takes a `const char*` to create a `Str`.

Constructors that take a single argument acts as **user-defined conversion**.

- The compiler will use this constructor to convert a `const char*` into a `Str`.
- The compiler uses the `Str(const char*)` constructor to create a temporary of type `Str`, than calls the appropriate (synthesized) operation to copy/assign this temporary.

Str operations

We want to be able to write things like:

```
s[i]
cin >> s
cout << s
s1 + s2
```

Indexing is easy: just add two member functions overloading the index operator:

```
char& operator[](size_type i) { return data[i]; }
const char& operator[](size_type i) const {
    return data[i];
}
```

Input-output operators

Input operator will change the state of the object, so it appears it should be a member function.

But . . .

Input-output operators

Input operator will change the state of the object, so it appears it should be a member function.

But . . .

```
cin >> s;
```

is equivalent to

```
cin.operator>>(s);
```

which calls the overloaded `>>` operator for the object `cin`.

But we do not own the definition of `istream` class, so we cannot add this overloaded operator!

Input-output operators (2)

But if we make `operator>>` a member of `Str`, then our users can only write

```
s >> cin;
```

which is equivalent to

```
s.operator>>(cin);
```

But that's not what we want!

So the input-output operators must be non-members. Add the following to `Str.hpp`.

```
std::istream& operator>>(std::istream&, Str&);
std::ostream& operator<<(std::ostream&, const Str&);
```

Output operator

```
ostream& operator<<(ostream& os, const Str& s) {
    for (Str::size_type i = 0; i != s.size(); ++i)
        os << s[i];
    return os;
}
```

Each time through the loop:

- Invoke `Str::operator[]` to fetch a character.
- Invoke the STL-provided `operator<<` that takes an `ostream` and a `char`.

Input operator

```cpp
istream& operator>>(istream& is, Str& s) {
    // Obliterate existing value(s)
    s.data.clear();
    // Read and discard leading blanks
    char c;
    while (is.get(c) && isspace(c))
        ; // Nothing to do, except testing the condition

    // If we read a non-blank char, continue to read
    // until next whitespace character
    if (is) {
        do { s.data.push_back(c); }
        while (is.get(c) && !isspace(c));
        // If we read a blank, put it back on the stream
        if (is)
            is.unget();
    }
    return is;
}
```

Input operator

This is all well, but the code on previous slide will not compile!

Why?

Input operator (2)

Previous code will not compile, because `operator>>` is not a member function, yet it tries to access `private` data member directly!

But it **needs** to do so, and we do not want our users to have direct access to our `private` members through the interface!

Solution: make ther `operator>>` a `friend` of the `Str` class:

```
class Str {
    friend std::istream& operator>>(std::istream&, Str&);
    // As before
};
```

Friends can access `private` members of the class.
`friend` declaration can appear anywhere in the class definition; but usually put near the `public` interface.

Other binary operators

Concatenation: + does not change the state of the concatenated object, so can be a non-member.
But compound concatenation += does change its left-hand side operand, so should be a member function:

```cpp
Str& operator+=(const Str& s) {
    std::copy(s.data.begin(), s.data.end(),
              std::back_inserter(data));
    return *this;
}
```

And then implement + using +=:

```cpp
Str operator+(const Str& lhs, const Str& rhs) {
    Str res = lhs; // Copy constructor
    res += rhs; // res.operator+=(rhs)
    return res; // Copy constructor*
}
```

Mixed-type expressions

```
Str greeting = "Hello, " + name + "!";
```

is equivalent to:

```
Str greeting = ("Hello, " + name) + "!";
```

Compiler will invoke conversion constructor on `const char*` and use Str `operator+`.

So the code is in fact equivalent to:

```
Str temp1("Hello, ");
Str temp2 = temp1 + name;
Str temp3("!");
Str greeting = temp2 + temp3;
```

In practice, to avoid cost of temps, one can define specific versions of operators (here concatenation) for every combination of operands (and not rely on automatic conversions).

Designing binary operators

Class membership is very important for conversions.

- If an operator is member of class, then left-hand operands cannot result from conversion – **must** be of class type.
    - Compiler only looks for non-member operators and for the corresponding class one.
- If operator is class member, then the operator is **asymmetrical**:
    - as right operand can be converted, but left-hand operand cannot;
    - Better to keep symmetry by making operator a non-member function.

Conversion hazards

- If a constructor defines the **structure** of the object constructed, *not* its **content**, the constructor should be **`explicit`**.
- If the constructor defines the content of the object, then it *should not* be **`explicit`**.
- Example:

```
vector<string> p = frame(42);
```

We get a `vector` with 42 empty rows. It is likely the user expected a framed number 42!

This single argument constructor is a structural constructor, so turn off automatic conversion by using **`explicit`** (as we did in lecture 6).

Conversion operators

We saw that some constructors allows conversion **to** the class type. **Conversion operators** define conversion **from** the class type.

Conversion operators must be class members and their name is `operator` followed by the target type name:

```
class Student_info {
public:
    operator double() const;
    // ...
};
```

would be used whenever a `Student_info` object is used where a `double` is expected.

```
vector<Student_info> students;
// Fill in students
double d = 0;
for (auto& student : students)
    d += student; // student implicitly converted to double
cout << "Average grade: " << d / students.size() << endl;
```

Conversion operators (2)

`istream` uses conversion operators to support expressions such as:

```cpp
cin >> x;
if (cin) { /* ... */}
```

There is a conversion operator from `istream` to `void*`:

- Returns either `nullptr` or implementation-defined non-zero `void*`.
- `void*` is universal pointer, but **can't be dereferenced**.
- `void*` can be converted to a `bool`.

Why not support a direct conversion from `istream` to `bool`?

Conversion operators (2)

`istream` uses conversion operators to support expressions such as:

```
cin >> x;
if (cin) { /* ... */}
```

There is a conversion operator from `istream` to `void*`:

- Returns either `nullptr` or implementation-defined non-zero `void*`.
- `void*` is universal pointer, but **can't be dereferenced**.
- `void*` can be converted to a `bool`.

Why not support a direct conversion from `istream` to `bool`?
Suppose one writes the following erroneous code:

```
int x;
cin << x; // Should have written cin >> x;
```

`cin` would get converted to a `bool`, then an `int`, then shifted left by `x` bits, then result discarded!
On the other hand, `void*` cannot be converted to arithmetic value.

Conversion and memory management

Should `Str` support direct conversion to null-terminated array of characters?

```
operator char*();
operator const char*() const;
```

It turns out this is not a good idea:

- Can't return pointer to internal data: would violate encapsulation/protection.
- Even returning a `const` pointer does not work: object could be destroyed while pointer still alive.
- If we make a copy and return a pointer: implicit conversion creates copies, but never gives the user the corresponding pointer, so we can never reclaim the space!

```
Str s;
ifstream is(s); // Implicit conversion -- how can we free?
```

Conversion and memory management (2)

Solutions: use explicit member functions!

Example from STL (`string`):

- `c_str()` copies content of `string` into null-terminated **`char`** array:
  - The `string` class owns the array.
  - User expected **not** to delete it.
  - Data in array is ephemeral:
    - Only valid until next call to member function that might change the `string`.
    - Users expected to either use pointer immediately or copy content to user managed storage.
- `data()` same as `c_str()` but not null-terminated.
- `copy` takes a **`char`**`*` and an **`int`**: copies as many characters as indicated by **`int`** into space pointed to by **`char`**`*`.

Note that `c_str()` and `data()` can still point into destroyed object, but reduced risk as calls are **explicit**.

Use **smart pointers** to explicitly give ownership back to caller.