Programming with abstract data types

Solution to Assignment 1

What do we need?

- A `Token` class to represent strings or line breaks
- A `Line` class to represent a growing line of text
- A program to reformat lines of text
- Glue to hold everything together

Defining classes

- We define the class itself:

```
class Foo {
public:
        // interface
private:
        // implementation
};
```

access labels

- We then define the functions that constitute the implementation

The Token class

constructor

member functions

```
class Token {
public:
    Token(FILE*);
    Toktype type();
    string word();
private:
    // …
};
```

How will we implement it?

- Every token knows what type it is
- If a token represents a word, it will contain a string
- For now, let's keep it simple; we will revisit this strategy later

Representing the kind of token

- Enumerated types are a convenient way of using names instead of magic numbers

```
enum Toktype {
    WORD, BREAK, END
};
```

Finishing the definition

```
class Token {
public:
    Token(FILE*);
    Toktype type();
    string word();
private:
    string s;
    Toktype t;
};
```

Defining the member functions

```
Toktype Token::type()
{
    return t;
}
string Token::word()
{
    assert(type() == WORD);
    return s;
}
```

Now comes the (first) hard part

- Reading a Token from an input file requires us to know how input works
  - `getc` will return a character or EOF
  - `feof` will tell us if a previous call to `getc` returned EOF
  - `ungetc` will let us push a single character back into the input, and will do nothing if we try to push EOF

Token constructor, part 1

```
Token::Token(FILE* f)
{
    if (feof(f))
        t = END;
    else {
        // We have work to do
    }
}
```

What are the possibilities?

- White-space characters that include two or more newlines (we return BREAK)
- Zero or more white-space characters including at most one newline, then
  - a non-space character (we must read characters and return WORD), or
  - end of file (we must return END)
- In all cases, we will read one extra char

Our first task

- Read as many white-space characters as we can, remembering the first character (or EOF) afterwards, and counting newlines
- We will rely on the `isspace` function from the standard library

Scanning white-space characters

This type must be int, not char, so that we can hold any character and still have room for EOF

```
int c = getc(f);
int nl = 0;
while (isspace(c)) {
    if (c == '\n')
        ++nl;
    c = getc(f);
}
```

Where are we now?

- We have just read something that is not white space; it could be
  - the first character of a word, or
  - EOF
- If we have seen >=2 newlines, we want to defer (i.e. push back) whatever comes next

Dealing with what we scanned

```
if (nl >= 2) {
    t = BREAK;
    ungetc(c, f);
} else if (c == EOF)
    t = END;
else {
    // still more work to do
}
```

Reading a word: strategy

- We know we have a word because we did not find a paragraph break and we have read a non-space character
- We must accumulate characters until we find white space or EOF
- We will have read one character too far, so we must put it back

Reading a word: code

```
T = WORD;
do {
    s += char(c);
    c = getc(f);
} while
   (c != EOF && !isspace(c));
ungetc(c, f);
```

A note on `ungetc`

- Every alternative ended with a call to `ungetc` except the case where we have read EOF
- We know that `ungetc(EOF, f)` has no effect
- We can therefore merge the calls to `ungetc` into one

The reduced code

```
if (nl >= 2)
    t = BREAK;
else if (c == EOF)
    t = END;
else {
    t = WORD;
    do {
        s += char(c);
        c = getc(f);
    } while (c != EOF && !isspace(c));
}
ungetc(c, f);
```

The `Line` class

```
class Line {
public:
    Line(int);
    void reset();
    bool canfit(string);
    void append(string);
    void print(FILE*);

private:
    string w;
    int max;
};
```

The `Line` constructor

```
Line::Line(int n):
    max(n) {  }
```

constructor initializer

nothing else to do

The `reset` function

```
void Line::reset()
{
   w = "";
}
```

The `canfit` function

```
bool Line::canfit(string s)
{
    if (w.empty())
        return s.length() <= max;
    return (
     w.length() + s.length() + 1
       ) <= max;
}
```

The `append` and `print` functions

```
void Line::append(string s)
{
    if (!w.empty())
        w += " ";
    w += s;
}
void Line::print(FILE* f)
{
    if (!w.empty())
        fprintf(f, "%s\n", w.c_str());
}
```

Putting it all together

```
void reformat(FILE* in, FILE* out, int width) {
    Line l(width); Token t(in);

    while (t.type() != END) {
        if (t.type() == BREAK) {
            l.print(out); l.reset();
            fprintf(out, "\n");
        } else {
            if (!l.canfit(t.word())) {
                l.print(out); l.reset();
            }
            l.append(t.word());
        }
        t = Token(in);
    }
    l.print(out);
}
```

And finally, the main program...

```
int main()
{
    reformat(stdin, stdout, 60);
    return 0;
}
```

Thoughts on the program

- We introduced the notion of an EOF token during implementation
- It can still bear improvement…
- …but it is reasonably abstract, and reasonably straightforward

Dependencies

- The I/O library
- The `string` class
- The `isspace` library function

Mechanics

- The complete code is linked from the course home page
- I have compiled it under g++ 2.8.0, which I have asked to be installed on the CS machines
- It will not compile under g++ 2.7, which is more than two years old
- More details on Monday