


# Programming with abstract data types

A decorative graphic at the bottom of the slide consists of a red trapezoidal shape on the left and a black trapezoidal shape on the right. The red shape is a right-angled triangle with its hypotenuse sloping downwards from left to right. The black shape is a larger trapezoid that starts from the right side of the red shape and extends to the right edge of the slide, sloping upwards from left to right.

# Where are we now?

- We have a preliminary design for a line-breaking program
- That program uses line and token abstractions, which we must implement
- We will do it in C++

# 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 C++ classes

- We define the class itself:

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

- We then define the functions that constitute the implementation

# The Token class

```
class Token {  
public:
```

```
    Token(istream*);  
    Toktype type() const;  
    string word() const;
```

```
private:
```

```
    // ...  
};
```

*constructor*

*member functions*

*Saying const here is  
a promise that calling this  
function will not change the  
value of the object*

# How will we implement it?

- Every token knows what type it is
- If a token represents a word, it will contain a string
- This strategy is somewhat simplistic, so we will revisit it 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(istream*);  
    Toktype type() const;  
    string word() const;  
private:  
    string s;  
    Toktype t;  
};
```



# Defining the member functions

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

# Now comes the (first) hard part

- Reading a Token from an input stream requires us to know how input works
  - `get()` will return a character or EOF
  - `eof()` will tell us if a previous call to `get` returned EOF
  - `putback(c)` will let us push a single character back into the input; we must not try to push EOF
- These functions are all members of `istream`

# Token constructor, part 1

```
Token::Token(istream* i)
{
    if (i->eof())
        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) to tell if a character is white space

# Scanning white-space characters

```
int c = i->get();  
int nl = 0;  
while (isspace(c)) {  
    if (c == '\n')  
        ++nl;  
    c = i->get();  
}
```

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

# 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  $\geq 2$  newlines (i.e., if  $n \geq 2$ ), we want to defer (push back) whatever comes next

# Dealing with what we scanned

```
if (n1 >= 2) {  
    t = BREAK;  
    if (c != EOF)  
        i->putback(c);  
} 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 = i->get();
} while (c != EOF &&
        !isspace(c));
if (c != EOF)
    i->putback(c);
```

# A note on putback

- Every alternative ended with a call to putback except the case where we have read EOF
- We can therefore merge the calls to putback into one, after checking that we are not trying to put back EOF.

# The reduced code

```
if (n1 >= 2)
    t = BREAK;
else if (c == EOF)
    t = END;
else {
    t = WORD;
    do {
        s += char(c);
        c = i->get();
    } while (c != EOF && !isspace(c));
}
if (c != EOF)
    i->putback(c);
```

# The Line class

```
class Line {  
public:  
    Line(int);  
    void reset();  
    bool canfit(string) const;  
    void append(string);  
    void print(ostream*) const;  
  
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) const
{
    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(ostream* o) const
{
    if (!w.empty())
        *o << w << endl;
}
```

# Putting it all together

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

    while (t.type() != END) {
        if (t.type() == BREAK) {
            l.print(out); l.reset();
            *out << endl;
        } 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(&cin, &cout, 60);
    return 0;
}
```

# Dependencies

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

# Putting it all together

- The easiest way:
  - An `#include` for each library facility
  - For each class:
    - the class definition
    - the member function definitions
  - Functions that are not part of any class
  - Finally, the `main` function
- Ideally, we should use multiple files

# Thoughts on the program

- We could improve the program...
- ...but it is reasonably abstract, and reasonably straightforward
- We introduced the notion of an END token during implementation, not during design

# Language dependencies

- Abstraction is rarely impossible in any language
  - It may require discipline on the part of users
  - There may be a price in convenience or efficiency
- In particular, we can implement this solution using C instead of C++

# Objects

- This program used objects of Line and Token types to contain the state of the program
- We often think of an operation such as append as “acting on” an object
- Such actions are a fundamental part of object-oriented programming



# Are objects necessary?

*No!!*

- To understand this claim better, let's review what objects are, and how they differ from abstract data types in general

# What is an object?

- It contains information
- There are operations defined on it
- It has identity
  - Two different objects can contain identical information, and support identical operations, but still be different objects

# What is object identity?

- Suppose  $x$  and  $y$  are names of objects.
- How can we tell if  $x$  and  $y$  are merely two different names for the same object?
  - Perhaps we can take the address of  $x$  and of  $y$
  - Perhaps we can change one of them and see if the change is reflected in the other

# Identity and mutable state

- Unless you have a way of checking the identity of an object directly (such as its address), the only way to determine if two objects are the same is to change one and see if the other changes
- Therefore immutable objects have no identity.
- Immutable objects are just values!

# Functional programming

- There is a school of thought that says that mutable objects are evil, and all computation should be done with pure values
  - usually called *functional programming*
  - major languages: ML, Haskell
  - very impressive results in the laboratory
  - limited commercial relevance so far

# The key issues

- Functional and object-oriented programming are styles of abstraction that address similar problems
  - How do we partition a program?
  - What information flows between the parts of the program?
  - What do authors of one part have to know about other parts?
- They come to very different conclusions

# Summary

- We have seen an object-oriented implementation of the line-breaking problem.
- Other languages offer different tools for abstraction
  - Every language offers something
  - No language offers everything
- To use a language effectively, you must match its tools to the problem at hand

# Homework (due Monday)

- Start with an implementation of the line-filling program. You can use ours or write your own
  - Note: Our version needs egcs or g++ 2.8
- Modify it to put its output in multiple columns, and to make all output lines (except for the last line of a paragraph) exactly the same length by expanding spaces (approximately) equally