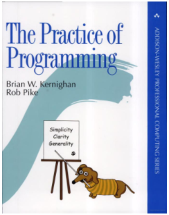
Interfaces[1]



[1] Examples in these slides come from Brian Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999

Intro

Interfaces

> *The essence of design is to balance competing goals and constraints. Although there are many tradeoffs when one is writing a small self-contained system, the ramifications of particular choices remain within the system and affect only the individual programmer. But when code is to be used by others, decisions have wider repercussions.*

Objective: To discuss the considerations that must be addressed when designing an interface. To illustrate these design issues with a simple yet useful example.

The Interface

- Interface is a contract between the supplier (programmer) and the client (user) (often another programmer)
- It describes what services (behaviors) and accesses are offered
- An interface should describe what it does, might do, and any changes it might make

Themes

Services

What services and accesses are provided?

- Provide enough functionality to be easy to use
- The interface should not become unwieldy
- Services should be uniform and convenient

Information Hiding

What information is visible?

- Interface should provide straightforward access to the components
- Interface should hide implementation details, so they can be changed without affecting clients

Resource Management

Who is responsible for managing memory and other resources

- Examples include:
  - Allocating and freeing storage
  - Managing shared memory
  - Open files
- A very general rule of thumb is that the person who makes the mess cleans it up
- This is, at times, not workable
- Interface must inform clients of expectations and responsibilities

Error Handling

Part of creating an interface is deciding how to signal errors

- Errors are typically detected low, handled high
- Errors can be signalled in various ways
  - Set a global value
  - Return an error code
  - Throw an exception[1]
- A very general rule of thumb is that the person who makes the mess cleans it up
- This is not often workable
- Interface must inform clients of expectations and responsibilities

[1] Use exceptions only for exceptional situations

Recap

(General) principles of a good interface:

- Hide implementation details

Recap

(General) principles of a good interface:

- Hide implementation details
- Choose a small, orthogonal set of primitives

Recap

(General) principles of a good interface:

- Hide implementation details
- Choose a small, orthogonal set of primitives
- Don't reach behind the user's back

Recap

(General) principles of a good interface:

- Hide implementation details
- Choose a small, orthogonal set of primitives
- Don't reach behind the user's back
- Do the same thing the same way everywhere

Recap

(General) principles of a good interface:

- Hide implementation details
- Choose a small, orthogonal set of primitives
- Don't reach behind the user's back
- Do the same thing the same way everywhere

Two other topics that are discussed when designing an interface:

Recap

(General) principles of a good interface:

- Hide implementation details
- Choose a small, orthogonal set of primitives
- Don't reach behind the user's back
- Do the same thing the same way everywhere

Two other topics that are discussed when designing an interface:

- Resource Management

Recap

(General) principles of a good interface:

- Hide implementation details
- Choose a small, orthogonal set of primitives
- Don't reach behind the user's back
- Do the same thing the same way everywhere

Two other topics that are discussed when designing an interface:

- Resource Management
- Error Handling

Principles

Sufficient

Hide implementation details

- You want to provide enough functionality
  - Clients should not need to open the hood
- Separating the interface from the implementation, minimising interdepencies, is important
  - Modular pieces are easier to test, debug
  - Easier to replace
  - Modular pieces improve portability

Necessary

Small, orthogonal set of primitives

- *Orthogonal:* non-overlapping functions, capabilities[1]
- Necessary, minimum set of behaviors
- Larger interfaces are harder to master
- Larger interfaces are harder to maintain

[1] These are general guidelines. Consider a GUI, as a counter-example

No Surprises

Don’t reach behind the user’s back

- Minimise side effects
- Document any side effects
- Decrease coupling
- Explicitly define what external services are needed

Consistent

Do the same thing the same way everywhere

- Use consistent naming schemes
- Define arguments consistently
  - E.g., order of target and source
- Behave similarly to associated interfaces
  - E.g., `mem*` and `str*` functions

Working Example

A CSV Library

We will create a C library that parses CSV files

- Each line of text is a record
- Fields are separated by commas
- Format widely used by spreadsheets and others
- E.g.:

```
"Reader",CS265,"Programming Tools and Techniques",A
"Slacker",CS265,"Programming Tools and Techniques",
```

Create a Prototype First

Plan to throw one away; you will, anyhow. – Frederick Brooks

It is not usually until you've built and used a version of the program that you understand the issues well enough to get the design right.

First version:

- Ignore many of the difficulties
- Complete enough to be useful and to gain some familiarity with the problem

Benefits of a Prototype

- You'll get a quick overview of the problem
- You'll get a feel for the larger difficulties
- Pin down the specifications
  - You'll know more/better questions to ask the client
  - You can have the client OK the behavior, and the look/feel
- Gives you a chance to check your assumptions

C Prototype, pg. I


```
char buf[200] ;        /* input line buffer */
char* field[20] ;      /* fields */

int csvgetline( FILE *fin )
{
   int nfield ;
   char *p, *q ;

   if( fgets( buf, sizeof(buf), fin )==NULL )
      return -1 ;

   nfield = 0 ;
   for( q=buf; (p=strtok( q, ",\n\r" ))!=NULL; q=NULL )
      field[nfield++] = unquote(p) ;

   return nfield ;
}
```

C Prototype, pg. 2


```
   /* remove leading and trailing quote */
char* unquote(char *p)
{
   if( p[0]=='"' ) {
      if( p[strlen(p)-1]=='"' )
         p[strlen(p)-1] = '\0' ;
      p++ ;
   }
   return p ;
}

int main( void )
{
   int i, nf ;
   while( (nf=csvgetline( stdin )) != -1 )
      for( i=0; i<nf; i++ )
         printf( "field[%d] = %s\n", i, field[i] ) ;
   return 0 ;
}
```

Decisions Made in Prototype, pg. 1

The decisions (and assumptions), right or wrong, that were made:

- Doesn't handle lines more than 199 characters
- Assumes records have at most 20 fields
- Records separated by newline
- Fields are separated by commas
- Surrounding quotes removed – no embedded quotes
- Input line not preserved – overwritten when creating fields
- No data saved from one input line to the next

Decisions Made in Prototype, pg. 2

- Did **not** separate implementation from interface
  - Access to fields through global variable (does not prevent access beyond last field)
- Global variables make code unsuitable to multi-threaded environment or interleaved calls
  - Can't parse 2 files simultaneously
- Caller must open and close files
- Input and splitting are inextricably linked

Each decision is interwoven into the code. There is no way to change any of these properties without changing the code

Specifications

Now we have some familiarity with the problem domain, we can ask soem good questions. Here's what was decided:

- Fields are separated by commas
- A field may be enclosed in double-quotes
- A quoted field may contain commas but not newlines
- A quoted field may contain double-quotes, represented by `""`
- Fields may be empty; `""` and empty string both represent an empty field
- Leading and trailing white space is preserved

Specifications – `csvgetline`


```
char* csvgetline( FILE* f ) ;
```

- Reads one line from open input file `f`
  - Assumes that input lines are terminated by `\r` `\n` `\r\n` `EOF`
- Returns pointer to line, with terminator removed, or `NULL` if EOF occurred
- Line may be of arbitrary length; returns `NULL` if memory limit exceeded
- Line must be treated as read-only storage
  - Caller must make a copy to preserve or change contents

Specifications – `csvnfield`


```
int csvnfield( void ) ;
```

- Returns number of fields on last line read by csvgetline
- Behavior undefined if called before csvgetline is called

Specifications – `csvfield`


```
char* csvfield( int n ) ;
```

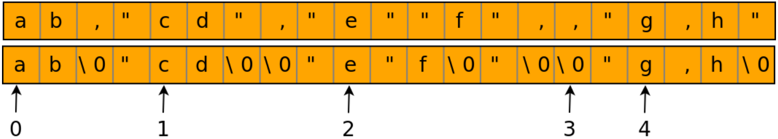
- Returns n-th field from last line read by csvgetline
  - Fields are numbered from 0.
  - Returns NULL if n<0 or beyond last field.
- There can be an arbitrary number of fields of any length;
- Caller must make a copy to preserve or change contents
- Behavior undefined if called before csvgetline is called

Specifications – `csvfield` (cont.)

- Fields must be treated as read-only storage
- Fields are separated by commas
- Fields may be surrounded by `"  "`
  - Such quotes are removed
  - Within `"  "`, `""` is replaced by `"`
  - Within `"  "`, comma is not a separator
- In unquoted fields, quotes are regular characters
- Returns NULL if memory limit exceeded

Example

```
$ gcc csvgetline2.c
$ ./a.out <<< 'ab,"cd","e""f",,"g,h"'
line = ab,"cd","e""f",,"g,h"
field[0] = ab
field[1] = cd
field[2] = e"f
field[3] =
field[4] = g,h
```



Decisions Made in Revision

Per the specs, these assumptions are okay, but, should be noted

- Records separated by newline
- Fields are separated by commas
- No data saved from one input line to the next
- Global variables make code unsuitable to multi-threaded environment or interleaved calls
- Caller must open and close files

Encapsulation

CSV Library in C++ — `csvgetline.cc`

- Much of the CSV machinery is the same
- Many of the details (resizing arrays) are hidden behind the `string` and `vector` interfaces
- Not a requirement, but, CSV fields are now wrapped in an object
  - We can have multiple CSV objects, parse different files concurrently
  - This is called *encapsulation*

Encapsulation in the C Version

- We've already achieved data hiding
- We can also encapsulate those global fields
- Each function will take a pointer to the data
  - Much like C++'s implicit `this` pointer
- We need an `init` method, to emulate the constructor
- We'll also need a destructor. Maybe the `reset` function
- Since C allows forward declarations of struct types, we can even moved the struct definition into the implementation file

Encapsulation – csv.h

csv.h

```
#ifndef __CSV_KP_
#define __CSV_KP_

struct csv_s {
   FILE *fin ;
   char *line , *sline ;
   int  maxline ;
   char **field ;
   int  maxfield , nfield ;
} ;

typedef csv_s* Csv ;

Csv   csv_init( FILE* ) ;
char* getline( Csv ) ;
char* field( Csv, int ) ;
int   nfield( Csv ) ;
void  kill( Csv ) ;

#endif /* __CSV_KP_ */
```

Encapsulation – csv_main.h

So, the client can use the CSV library, completely unaware of the underlying details:

csv.main

```
#include "csv.h"

int main( void )
{
   int i ;
   char *line ;
   Csv cf = csv_init( stdin ) ;

   while( (line=getline( cf ))!=NULL ) {
      printf( "line = %s\n", line ) ;
      for( i=0; i<nfield( cf ); ++i )
         printf( "field[%d] = %s\n", i, field( cf, i )) ;
      printf( "\n" ) ;
   }
   return 0 ;
}
```

Recap

(General) principles of a good interface:

- Hide implementation details
- Choose a small, orthogonal set of primitives
- Don't reach behind the user's back
- Do the same thing the same way everywhere

Two other topics that are discussed when designing an interface:

- Resource Management
- Error Handling