

Interface issues

- **interface: detailed boundary between code that provides a service and code that uses it**
- **functionality**
 - features and operations provided
 - inputs and return values
- **information hiding**
 - what parts of implementation are visible
 - what are hidden
- **resource management**
 - creation and initialization
 - maintaining state
 - ownership: sharing and copying
 - memory management
 - cleanup
- **error handling**
 - what errors are detected?
 - how are they handled or reported?
- **other issues**
 - orthogonality, convenience, simplicity, generality, consistency, regularity, motherhood, apple pie, ...

C interface for an RE package

- **functions from assignment 1:**

```
RE *RE_compile(char *)
int RE_match(RE *, char *)
char *RE_start(RE *)
char *RE_end(RE *)
void RE_free(RE *)
...
```

- **"RE" is an *opaque type***
 - conceals the implementation as much as possible
- **implementation uses a structure like this**

```
typedef struct RE {
    ...
} RE;
```
- **user code sees only**

```
typedef struct RE *RE;
```
- **analogous to FILE* in C stdio**
- **in real life, there would be a header file RE.h**

What functions?

- **relatively few**
- **fundamental, most commonly used**
- **not easily synthesized from others**
 - but others can be synthesized from them
- **not easily implemented by users**

- **this would be sufficient**

```
int RE(char *re, char *str,
        char **start, char **end);
```
- **but not really convenient or efficient**
- **typically compile once, test matches often**
- **often don't care about the matched string**

- **separate these into different functions**
 - compile a regexp (constructor)
 - match a string
 - access matched substring
 - free (destructor)

Convenience & usability issues

- **small things, but they make a difference**

- **should there be functions for common operations**
 - immediate match of a regexp and string
like Java's Pattern.matches(regexp, string)
(which is an anchored match!!)

- **which of these is best?**

```
char *RE_start(), char *RE_end()
char *RE_start(), int RE_length()
char *RE_matched_substr()
```

- **how should errors be reported and returned?**
 - bool, int, struct, pointers?
 - print? assertion failure?

- **consistency in choices, naming, order of args, ...**

More questions

- **when are the pointers `start()` and `end()` valid?**
 - what if source string changes?
 - what if multiple matches are in process?
- **what if you want successive searches, as in Java's `Matcher.find()`?**
 - who remembers where you were?
 - what if the source string has changed in the interim?
 - how do you make it re-entrant?
 - why is C's `strtok` is a botch?
- **what if there were an array of matched substrings?**
 - like Perl's `$1`, `$2`, ...
- **suppose RE's were to be cached as in Awk**
 - how are they coordinated?
- **how would you know if the RE had changed?**
 - is the string saved? hashed? quietly assumed ok?

Who manages what memory when?

- **a big, fundamental interface issue**
 - getting it wrong or inconsistent is a major problem
 - making it hard for users is a major problem
- **`char *RE_substr(RE *)` needs space for string**
- **who allocates space for the string?**
- **should it grow? without limit?**
- **who grows it?**
- **who complains if it gets too big? how?**
- **who owns it?**
- **who can change its contents? how?**
- **who sees the changes? re-entrant?**
- **what is its lifetime?**
 - when are pointers into the data structure invalidated?
- **who frees it?**
- **these issues are not all solved by garbage collection**
- **one specific choice: interface owns the storage**
- **rule for client: "don't change, don't free"**
 - may not be right but it's easy to state and understand