Program Design

1. Problem statement and requirements:
   What is the problem?

2. Specification:
   Detailed description of what the system does instead of how.

3. Design:
   Explore design space (like “back of the envelope” calculations), identify algorithms and key interfaces

4. Programming:
   Implement it in the simplest possible way; use libraries

5. Testing:
   Debug and test until the implementation is correct

6. Iterate:
   Do the design and implementation conform to the specification?
Stepwise Refinement

- Top-down design
  starts with a *high-level abstract* solution
  refines it repeatedly by successive transformations to lower-level solutions
  refinement ends at programming language statements
- Key idea: each refinement or *elaboration*
  must be *small, and correct*
  must move toward final solution
- Accompany refinements with *assertions* to help ensure *correctness*
- Refinements use English and pseudocode, but ultimately result in *code*:

```
English/pseudocode
    \downarrow
    C code
```
Example: How Many Library Books are Never Used?

1. Problem statement:
   The circulation file has a line of author & title for each checked out book.
   Need a program to answer how many books circulate in a year

2. Specification:
   unique reads its standard input and prints the number of distinct (non-redundant) lines on the standard output

3. Design: how many unique lines are in a typical circulation file?
   top-down design
   "chunks" are pseudocode to be elaborated
   \[
   \textit{unique} \equiv \\
   \text{<for each line of input>}
   \text{<add the line to the set of strings>}
   \text{<count how many lines are in the set>}
   \text{<print the output>}
   \]

4. Programming: make forward progress by elaborating chunks
   \[
   \text{<count how many lines in the set> } \equiv \\
   \text{count } = 0;
   \text{<for each element of the set>}
   \text{count}++;\]
What Modules?

- ADTs: sets of strings

- Modules:

  main.c  handle command-line arguments (if any) and top-level loops

  <unique>  
  <includes>  
  <defines>
  int main(int argc, char *argv[]) {
    <locals>
    <for each line of input>
      <add the line to the set of strings>
    <count how many lines are in the set>
    <print the output>
    return EXIT_SUCCESS;
  }

  strset.h  interface for sets of strings
  strset.c  initial implementation of sets of strings

- Use RCS to track changes

  main.c,v
  strset.h,v
  strset.c,v
Elaboration

- Do the easy chunks first

  `<print the output> ≡`
  ```c
  printf("%d\n", count);
  ```

  `<locals> ≡`
  ```c
  int count = 0;
  ```

  `<includes> ≡`
  ```c
  #include <stdio.h>
  ```

- Some elaborations can be done `without` defining the ADTs

  `<for each line in the input> ≡`
  ```c
  while (gets(line))
  ```

  `<defines> ≡`
  ```c
  #define MAXLINE 512
  ```

  `<locals> +⇒` indicates that code is `appended` to the chunk
  ```c
  char line[MAXLINE];
  ```
ADT: Sets of Strings

strset.h describes abstract operations, not implementation; what, not how

#ifndef STRSET_INCLUDED
#define STRSET_INCLUDED

define T Strset_T
typedef struct T *T;

t T Strset_new(void); /* allocates and returns a new, empty set */
void Strset_free(T *set); /* deallocates *set and its contents, set *set to NULL */
void Strset_add(T set, char *str); /* adds str to set, if str is not already in set */
void Strset_delete(T set, char *str); /* removes str from set, if str is in set */
int Strset_member(T set, char *str); /* returns 1 if str is in set, else 0 */
void Strset_foreach(T set, void apply(char *str, void *cl), void *cl); /* executes apply(s, cl) for each string s in set */

/* It is a checked runtime error to pass a NULL T, *T, char*, or apply to any function in this interface. */

#undef T
#endif

opaque pointer type; clients can’t see innards

closure

client responsibilities
Elaboration, cont’d

• ADT interface gives enough information to finish the client, `main.c`

  `<locals>`
  ```c
  Strset_T set = Strset_new();
  ```

  `<includes>`
  ```c
  #include "strset.h"
  ```

  `<add the line to the set of strings>`
  ```c
  Strset_add(set, line);
  ```

  `<count how many lines are in the set>`
  ```c
  Strset_foreach(set, cardinality, &count);
  ```

  ```c
  static void cardinality(char *str, void *cl) {
      int *p = cl;
      (*p)++;
      /* or *(int *)cl++; */
  }
  ```

• Implement clients of ADTs *before* the ADTs themselves; helps expose design *inadequacies*
Strset

- Initial implementation can be *simple*; it might suffice ...

- Implementation *reveals* the innards of the *opaque* type: a list of strings

```
#include "strset.h"
#define T Strset_T
struct T {
    T next;
    char str[1];
};
```

- `Strset_new` allocates a new header node

```
T Strset_new(void) {
    T set = calloc(1, sizeof *set);
    assert(set);
    return set;
}
```

OK during development and in COS 217, but not in production programs
Initial Implementation of Strset

- For now, implement only enough of the ADT to test `unique`

```c
void Strset_add(T set, char *str) {
    T p = set;

    assert(set);
    assert(str);
    while ((p = p->next) != NULL)
        if (strcmp(str, p->str) == 0)
            return;
    p = malloc(sizeof *p + strlen(str));
    assert(p);
    strcpy(p->str, str);
    p->next = set->next;
    set->next = p;
}

void Strset_foreach(T set, void apply(char *str, void *cl),
                    void *cl) {
    assert(set);
    assert(apply);
    while ((set = set->next) != NULL)
        apply(set->str, cl);
}
```
5. Testing: `unique` works, but runs too slowly on `large` inputs; why?
   improve `strset`'s implementation; don't change its interface

- Solution: use a `hash table` to represent a set of strings
  a set is a pointer to an array of `TABLESIZE` linked lists
  crunch the string into an integer `h`
  let \( i = h \% TABLESIZE \)
  search the `i`th linked list for the string, or
  add the string to the head of the `i`th list

![Diagram of hash table]
Better Implementation of Strset

```c
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include "strset.h"
#define T Strset_T
#define TABLESIZE 97
struct T {
    struct elem {
        struct elem *next;
        char str[1];
    } *table[TABLESIZE];
};

void Strset_free(T *set) {
    int i;

    assert(set && *set);
    for (i = 0; i < TABLESIZE; i++) {
        struct elem *p, *q;
        for (p = (*set)->table[i]; p; p = q) {
            q = p->next;
            free(p);
        }
    }
    free(*set);
    *set = NULL;
}

T Strset_new(void) {
    T set = calloc(1, sizeof *set);
    assert(set);
    return set;
}
```

same as above!
Better Implementation of Strset, cont’d

```
static unsigned hash(char *str) {
    unsigned h = 0;

    while (*str)
        h = (h<<1) + *str++;
    return h;
}

void Strset_add(T set, char *str) {
    int i;
    struct elem *p;

    assert(set);
    assert(str);
    i = hash(str)%TABLESIZE;
    for (p = set->table[i]; p; p = p->next)
        if (strcmp(str, p->str) == 0)
            return;
    p = malloc(sizeof *p + strlen(str));
    assert(p);
    strcpy(p->str, str);
    p->next = set->table[i];
    set->table[i] = p;
}
```
void Strset_foreach(T set, void apply(char *str, void *cl),
void *cl) {
    int i;

    assert(set);
    assert(apply);
    for (i = 0; i < TABLESIZE; i++) {
        struct elem *p;
        for (p = set->table[i]; p; p = p->next)
            apply(p->str, cl);
    }
}

- see files in src/{strset, unique}; RCS files track all improvements
More Testing

- **More** testing
  - test on “typical” inputs
  - test on **extreme** inputs:
    - a file with blank lines
    - a very long file
    - a long file with lines that are all identical
    - a file with very long lines
    - an empty file
    - ...

- Very long lines causes **unique** to crash!

  \[
  \text{<for each line in the input>} \equiv \\
  \text{while (gets(line))}
  \]

  \text{gets can’t check length of line}

6. Iterate

  go to step 2, amend the **specification**:
  “Only the first 511 characters of a line are significant”

  go to step 4 (programming) and fix the error (use RCS)

  go to step 5 (testing) and repeat **all** of the tests

  \text{iterate} again.