CS360 Lecture Notes -- Fields

- James S. Plank

- Directory: **~plank/cs360/notes/Fields**
- Lecture notes: **http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Fields**

- Original notes: Wed Aug 25 11:01:12 EDT 1999
- Last Revision: Tue Jan 22 14:09:27 EST 2013

The fields library is a suite of routines that make reading input easier than using **getchar()**, **scanf()** or **gets()**. This is a library that I wrote -- it is not standard in Unix, but it should work with any C compiler (this includes on DOS/Windows). If you want to take the fields library with you after class, go ahead and do so. You can get the source code at **http://www.cs.utk.edu/~plank/plank/fields/fields.html**.

In order to use the fields procedures in this class, you should include the file **fields.h**, which can be found in the directory **/home/plank/cs360/include**. Instead of including the full path name in your C file, just do:

```
#include "fields.h"
```

and then compile the program with:

```
gcc -I/home/plank/cs360/include
```

When you link your object files to make an executable, you need to follow the directions in the Libfdr notes.

The **makefile** in this directory does both of these things for you. When you look over the file **printwords.c**, make sure you figure out how to compile it so that it finds **fields.h**, and so that the compilation links with **libfdr.a**.

The fields library defines and implements a data structure that simplifies input processing in C. The data structure consists of a type definition and four procedure calls. All are defined in **fields.h**:

```
#include <stdio.h>
#define MAXLEN 1001
#define MAXFIELDS 1000

typedef struct inputstruct {
  char *name;               /* File name */
  FILE *f;                  /* File descriptor */
  int line;                 /* Line number */
  char text1[MAXLEN];       /* The line */
  char text2[MAXLEN];       /* Working -- contains fields */
```

```
  int NF;                   /* Number of fields */
  char *fields[MAXFIELDS];  /* Pointers to fields */
  int file;                 /* 1 for file, 0 for popen */
} *IS;

extern IS new_inputstruct(/* FILENAME -- NULL for stdin */);
extern IS pipe_inputstruct(/* COMMAND -- NULL for stdin */);
extern int get_line(/* IS */); /* returns NF, or -1 on EOF.  Does not close
the file */
extern void jettison_inputstruct(/* IS */);  /* frees the IS and fcloses the
file */
```

To read a file with the fields library, you call **new_inputstruct()** with the proper filename. **New_inputstruct()** takes the file name as its argument (**NULL** for standard input), and returns an **IS** as a result. Note that the **IS** is a pointer to a **struct inputstruct**. This is **malloc()**'d for you in the **new_inputstruct()** call.
If **new_inputstruct()** cannot open the file, it will return **NULL**, and you can call **perror()** to print out the reason for the failure (read the man page on **perror()** if you want to learn about it).

Once you have an **IS**, you call **get_line()** on it to read a line. **Get_line()** changes the state of the **IS** to reflect the reading of the line. Specifically:

- It puts the contents of the line in **text1**.
- It breaks up the line into words. The **NF** field contains the number of words in the field. The first **NF** slots of the **fields** array point to each of the **NF** words (and these words are null-terminated).
- The **line** field contains the line number of the line.
- **Get_line()** returns the **NF** field as its return value.
- It returns -1 when it reaches the end of the file.

Jettison_inputstruct() closes the file associated with the **IS** and deallocates (frees) the **IS**. Do not worry about **pipe_inputstruct()** for now.

These procedures are very convenient for processing input files. For example, the following program (in **printwords.c**) prints out every word of an input file prepended with its line number:

```
#include <stdio.h>
#include <stdlib.h>
#include "fields.h"

main(int argc, char **argv)
{
  IS is;
  int i;
```

```
  if (argc != 2) {
    fprintf(stderr, "usage: printwords filename\n");
    exit(1);
  }

  is = new_inputstruct(argv[1]);
  if (is == NULL) {
    perror(argv[1]);
    exit(1);
  }

  while(get_line(is) >= 0) {
    for (i = 0; i < is->NF; i++) {
      printf("%d: %s\n", is->line, is->fields[i]);
    }
  }

  jettison_inputstruct(is);
  exit(0);
}
```

So, for example, if the file **rex-1.txt** contains the following three lines:

```
June: Hi ... I missed you!
Rex:  Same here!  You're all I could think about!
June: I was?
```

Then running printwords on rex-1.txt results in the following output:

```
UNIX> printwords rex-1.txt
1: June:
1: Hi
1: ...
1: I
1: missed
1: you!
2: Rex:
2: Same
2: here!
2: You're
2: all
2: I
2: could
2: think
2: about!
3: June:
3: I
3: was?
UNIX>
```

One important thing to note about **fields.o** is
that *only* **new_inputstruct()** calls **malloc()**. **Get_line()** simply fills in the fields of the **IS** structure --- it does *not* perform memory allocation. Therefore, suppose you wanted to print out the first word on the second-to-last line. The following program (**badword.c**) would not work:

```
#include <stdio.h>
#include <stdlib.h>
#include "fields.h"

main(int argc, char **argv)
{
  IS is;
  int i;
  char *penultimate_word;
  char *last_word;

  if (argc != 2) {
    fprintf(stderr, "usage: badword filename\n");
    exit(1);
  }

  is = new_inputstruct(argv[1]);
  if (is == NULL) {
    perror(argv[1]);
    exit(1);
  }

  penultimate_word = NULL;
  last_word = NULL;

  while(get_line(is) >= 0) {
    penultimate_word = last_word;
    if (is->NF > 0) {
      last_word = is->fields[0];
    } else {
      last_word = NULL;
    }
  }

  if (penultimate_word != NULL) printf("%s\n", penultimate_word);
  jettison_inputstruct(is);
  exit(0);
}
```

Why? Look at what happens when you execute it on rex-1.txt:

```
UNIX> badword rex-1.txt
June:
UNIX>
```

It prints ``June:" instead of ``Rex:" because **get_line()** does not allocate any new memory. Both **penultimate_word** and **last_word** end up pointing to the same thing.

Let's try another example which is probably a little more confusing:

```
UNIX> cat rex-2.txt
June: Hi ... I missed you!
  Rex:  Same here!  You're all I could think about!
June: I was?
UNIX> badword rex-2.txt
ne:
UNIX>
```

Remember, **get_line()** doesn't call **malloc()**. **Malloc()** is only called in **new_inputstruct()**. The **fields** pointers point to memory which is in the **text2** part of the struct. Thus, if the first word is the first character on the line, then **is->fields[0]** points to **is->text2**. If the first word is the third character on the line, then **is->fields[0]** points to **is->text2+2**. That is why the "ne:" is printed -- because**penultimate_line** points to **is->text2+2**. Since **is->text2** has been overwritten with **"June:"** in the **get_line()** call, **penultimate_line** points to "ne:".

Make sure you understand this example, because you can get yourself into a mess of trouble otherwise. If you're still having problems, put in some print statements and print out the addresses of the pointers.

The correct version of the program is in **goodword.c**: (note that this is a very inefficient program because of all the **strdup()** and **free()** calls. You could do better if you wanted to.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fields.h"

main(int argc, char **argv)
{
  IS is;
  int i;
  char *penultimate_word;
  char *last_word;

  if (argc != 2) {
    fprintf(stderr, "usage: badword filename\n");
    exit(1);
  }

  is = new_inputstruct(argv[1]);
  if (is == NULL) {
    perror(argv[1]);
    exit(1);
  }

  penultimate_word = NULL;
```

```
  last_word = NULL;

  while(get_line(is) >= 0) {
    if (penultimate_word != NULL) free(penultimate_word);
    penultimate_word = last_word;
    if (is->NF > 0) {
      last_word = strdup(is->fields[0]);
    } else {
      last_word = NULL;
    }
  }

  if (penultimate_word != NULL) printf("%s\n", penultimate_word);
  jettison_inputstruct(is);
  exit(0);
}
```

Field.o assumes that all input lines are less than 1000 characters.

---

tailanyf

Now, as another example, **tailanyf.c** takes *n* as a command line argument, and prints out the last *n* lines of standard input. It uses the fields library to read standard input.