CS360 Lecture Notes -- The Jval Type

- Jim Plank
- Directory: **~plank/cs360/notes/Jval**
- Lecture notes: **http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Jval**
- Tue Oct 13 10:07:44 EDT 1998

The Jval type

In **jval.h**, I define a type called a **Jval**. This is a big union:

```
typedef union {
    int i;
    long l;
    float f;
    double d;
    void *v;
    char *s;
    char c;
    unsigned char uc;
    short sh;
    unsigned short ush;
    unsigned int ui;
    int iarray[2];
    float farray[2];
    char carray[8];
    unsigned char ucarray[8];
  } Jval;
```

I make use of **Jval**'s when I write generic data structures such as lists, and trees. For the purposes of this class, don't worry about half of the fields in the union. The only important ones are:

```
    int i;
    float f;
    double d;
    void *v;
    char *s;
    char c;
```

The nice thing about a **Jval** is that you can hold one piece of data in a **Jval**, regardless of what the type of that piece of data is. Moreover, the **Jval** will always be 8 bytes. You use it just like the unions that were discussed in the union lecture notes from CS140. Take a minute to go over that lecture to brush yourself up on what a union is.

Constructor functions

You can of course, create and use a **Jval** by simply declaring and using it. For example:

```
  Jval j;

  j.i = 4;
```

You can freely pass **Jval**'s to and from procedure calls. A **Jval** can be the return value of a procedure call.

Jval.h defines a whole bunch of prototypes for ``constructor functions."

```
extern Jval new_jval_i(int);
extern Jval new_jval_f(float);
extern Jval new_jval_d(double);
extern Jval new_jval_v(void *);
extern Jval new_jval_s(char *);
```

These return **Jval**'s to you when you give them arguments of a specific type. For example, if you want to initialize a **Jval** so that it is in integer whose value is 4, you can do it as shown above, or you can do:

```
  Jval j;

  j = new_jval_i(4);
```

Now **j.i** will be the integer 4. You will see later (when we get to using some of the general purpose data structures) why this is convenient.

These ``constructor functions" are implemented in **jval.c**, and are very simple. For example, here is **new_jval_i()**:

```
Jval new_jval_i(int i) {
  Jval j;
  j.i = i;
  return j;
}
```

A simple example

Obviously, to use **Jval**'s, include **jval.h**, and then link with **libfdr.a** as described in the Libfdr lecture notes..

As an example, **jval_ex.c** is **union3.c** from the Union lecture, converted to use **Jval**s.

You'll note that there is only one change to the code: the typedef:

```
typedef struct {
  char type;
  Jval value;
} Item;
```

Since the **Jval** struct has the same **.i**, **.f** and **.s** fields, the rest of the code can remain unchanged.

As you can see, it works just fine:

```
UNIX> jval_ex
int 4
string Jim
float -33.2
```

```
int -2
int 1
Item 0: Type i -- Value: 4
Item 1: Type s -- Value: Jim
Item 2: Type f -- Value: -33.200001
Item 3: Type i -- Value: -2
Item 4: Type i -- Value: 1

Sizeof(Item): 16
UNIX>
```

In **jval_ex2.c**, we modify the code to use the constructor functions. The main change is (and I've bold-faced the new code):

```
  int i2;
  float f;

  ..

  for (i = 0; i < 5; i++) {
    if (get_line(is) != 2) exit(1);

    if (strcmp(is->fields[0], "int") == 0) {
      array[i].type = 'i';
      if (sscanf(is->fields[1], "%d", &i2) != 1) exit(1);
      array[i].value = new_jval_i(i2);

    } else if (strcmp(is->fields[0], "float") == 0) {
      array[i].type = 'f';
      if (sscanf(is->fields[1], "%f", &f) != 1) exit(1);
      array[i].value = new_jval_f(f);

    } else if (strcmp(is->fields[0], "string") == 0) {
      array[i].type = 's';
      array[i].value = new_jval_s(strdup(is->fields[1]));

    } else {
      exit(1);
    }
  }
```

A word of warning about Jval's

The purpose of the **Jval** type is to make general purpose data structures such as dllists and red-black trees as flexible and efficient as possible. You are *not* to use **Jval**'s in your code for any other reason. I will tell you when to use them.

Specifically, you are not to say, use a **Jval** instead of an **int** in your code just because it works. That makes your code unreadable, and unreadability is majorly bad. Here is an example of bad code to average all of the integers on standard input (in **badavg.c**):

```
main()
{
  Jval total;
  Jval j;
  Jval n;

  n.i = 0;
  total.i = 0;

  while (scanf("%d", &(j.i)) == 1) {
    total.i += j.i;
    n.i++;
  }

  total.d = ((double) total.i) / ((double) n.i);

  printf("Average = %lf\n", total.d);
}
```

Yes, it works, and yes, it's a cute way to use **total** as both an **int** and a **double**. But it is revolting -- every use of **Jval**'s is bad, and if you use them in ways like these, you will be punished.

(In case you care, the code should look as in **goodavg.c**):

```
main()
{
  int total;
  int j;
  int n;

  n = 0;
  total = 0;

  while (scanf("%d", &j) == 1) {
    total += j;
    n++;
  }

  if (n == 0) exit(1);

  printf("Average = %lf\n", ((double) total)/((double) n));
}
```

(You can do other things to make that casting look better too).

Accessor functions

I have put accessor functions into **jval.h**/**jval.c**. An accessor function simply lets you get the desired value out of a **Jval** by calling a function rather than accessing the field. Why would you want to do this? Well, like the constructor functions, it makes life easier in certain circumstances. The accessor functions are:

```
extern int    jval_i(Jval);
extern long   jval_l(Jval);
extern float  jval_f(Jval);
extern double jval_d(Jval);
extern void  *jval_v(Jval);
extern char  *jval_s(Jval);
extern char   jval_c(Jval);
...
```
So, for example, calling **jval_i(j)** is the same as using **j.i**.

JNULL

Finally, **jval.h** contains a global variable **JNULL**. Use this when you would use **NULL** for a **char *** or **void ***.