Fall 2006 Handout 9

Bit-fields: K&R pp. 149−150; King pp. 455−460

A *bit-field* is a field of a structure whose size is measured in bits. It can be less than one byte wide, and as small as a single bit. Make a bit-field when a structure contains several little numbers, and it would waste too much space to make each number an individual **`char`**.

Here is a one-bit integer capable of holding either 0 or 1, and a two-bit integer capable of holding either 0, 1, 2, or 3.

```
#include <stdio.h>
#include <stdlib.h>
#define N   10

typedef struct {
    double price;                  /* price of the rug, in dollars */
    unsigned int imported: 1;      /* 1 if imported, 0 if domestic */
    unsigned int instock: 2;       /* 2 in stock, 1 on order, 0 out of stock */
} rug_t;

int main()
{
    rug_t a[N];         /* an array of rugs */

    int i;
    unsigned temp;      /* can't give address of a bit-field to scanf */
    rug_t *p;

    for (i = 0; i < N; ++i) {
        printf("Please type the price of rug number %d: ", i);
        scanf("%lf", &a[i].price);       /* lowercase LF */

        printf("Type 1 if it's imported, 0 if it's domestic: ");
        scanf("%u", &temp);
        a[i].imported = temp;

        printf("Type 2 if in stock, 1 on order, 0 out of stock: ");
        scanf("%u", &temp);
        a[i].instock = temp;
    }

    printf("number\tprice\timported\tinstock\n");
    for (i = 0; i < N; ++i) {
        printf("%3d\t%7.2f\t%u\t%u\n", i,
            a[i].price, a[i].imported, a[i].instock);
    }

    for (p = a; p < a + N; ++p) {
        printf("Please type the price of rug number %d: ", p - a);
        scanf("%lf", &p->price); /* lowercase LF */

        printf("Type 1 if it’s imported, 0 if it’s domestic: ");
        scanf("%u", &temp);
        p->imported = temp;

        printf("Type 2 if in stock, 1 on order, 0 otherwise: ");
        scanf("%u", &temp);
        p->instock = temp;
    }

    printf("number\tprice\timported\tinstock\n");
    for (p = a; p < a + N; ++p) {
        printf("%3d\t%7.2f\t%u\t%u\n", p - a,
            p->price, p->imported, p->instock);
    }

    return EXIT_SUCCESS;
}
```

Unions: pp. 147−149

Secret Agent man
Secret Agent man—
They’ve given you a number
And taken ’way your name.

Some people have names, but some have only numbers. No one has both.

The complete program is not shown below. You must add the declaration and definition for the function `mygetline`.

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

#define N   10

typedef struct {
    char *name;      /* person’s name, or NULL if unused */
    int number;      /* person’s number, or undefined if unused */
} label_t;

int main()
{
    label_t a[N];
    int i;
    label_t *p;
    int which;           /* 1 for name, 0 for number */

    for (i = 0; i < N; ++i) {
        printf("Does this person have a name or a number?\n");
        printf("Type 1 for name, 0 for number: ");
        scanf("%d", &which);

        if (which == 1) {
            printf("Please type the name: ");
            a[i].name = mygetline();
        } else {
            printf("Please type the number: ");
            scanf("%d", &a[i].number);
            a[i].name = NULL;           /* unused */
        }
    }

    for (i = 0; i < N; ++i) {
        if (a[i].name == NULL) {
            printf("%d\n", a[i].number);
        } else {
            printf("%s\n", a[i].name);
            free(a[i].name);
        }
    }

    /* Another way to write the two for loops above. */
    for (p = a; p < a + N; ++p) {
        printf("Does this person have a name or a number?\n");
        printf("Type 1 for name, 0 for number: ");
        scanf("%d", &which);

        if (which == 1) {
            printf("Please type the name: ");
            p->name = mygetline();
        } else {
            printf("Please type the number: ");
            scanf("%d", &p->number);
            p->name = NULL;           /* unused */
        }
    }

    for (p = a; p < a + N; ++p) {
        if (p->name == NULL) {
            printf("%d\n", p->number);
        } else {
            printf("%s\n", p->name);
            free(p->name);
        }
    }

    return EXIT_SUCCESS;
}
```

Syntactically, a union is just like a structure: just change the word **`struct`** to **`union`** in the above program to make the array smaller.

The complete program is not shown below. You must add the declaration and definition for the function **`mygetline`**.

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

#define N   10
```

```c

typedef union {
    char *name;
    int number;
} label_t;

int main()
{
    label_t a[N];
    int which[N];    /* 1 if corresponding element in the a array contains a name,
                        0 if it contains a number */

    int i;
    label_t *p;
    int *q;

    for (i = 0; i < N; ++i) {
        printf("Does this person have a name or a number?\n");
        printf("Type 1 for name, 0 for number: ");
        scanf("%d", &which[i]);

        if (which[i] == 0) {
            printf("Please type the number: ");
            scanf("%d", &a[i].number);
        } else {
            printf("Please type the name: ");
            a[i].name = mygetline();
        }
    }

    for (i = 0; i < N; ++i) {
        if (which[i] == 0) {
            printf("%d\n", a[i].number);
        } else {
            printf("%s\n", a[i].name);
            free(a[i].name);
        }
    }

    /* Another way to write the two for loops above. */
    for (p = a, q = which; p < a + N; ++p, ++q) {
        printf("Does this person have a name or a number?\n");
        printf("Type 1 for name, 0 for number: ");
        scanf("%d", q);

        if (*q == 0) {
            printf("Please type the number: ");
            scanf("%d", p->number);
        } else {
            printf("Please type the name: ");
            p->name = mygetline();
        }
    }

    for (p = a, q = which; p < a + N; ++p, ++q) {
        if (*q == 0) {
            printf("%d\n", p->number);
        } else {
            printf("%s\n", p->name);
            free(p->name);
        }
    }

    return EXIT_SUCCESS;
}
```

A structure can contain another structure (p. 148):

```
typedef struct {
    int systolic;                   /* bigger number: contract */
    int diastolic;                  /* smaller number: expand */
} blood_pressure_t;

typedef struct {
    int pulse;                      /* heartbeats per minute */
    double temperature;             /* Fahrenheit */
    blood_pressure_t pressure;
    char insurance;                 /* 1 means insured, 0 means uninsured */
} signs_t;                          /* vital signs */

    signs_t a[] = {                 /* etc. */
    signs_t *p;

    int i;
    for (i = 0; i <                 /* etc. */
        printf("%d %d\n", a[i].pulse, a[i].pressure.systolic);

    blood_pressure_t p;
    for (p = a; p <                 /* etc. */
        printf("%d %d\n", p->pulse, p->pressure.systolic);
```

In fact, a structure can contain a union. This lets us combine the two parallel arrays **a** and **which** in the above program into a single array of structures.

The complete program is not shown below. You must add the declaration and definition for the function **mygetline**.

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

#define N   10

typedef union {
    char *name;
    int number;
} label_t;

typedef struct {
    int which;   /* 1 if lab contains name, 0 if lab contains number */
    label_t label;
} person_t;
```

```

int main()
{
    person a[N];
    int i;
    person *p;

    for (i = 0; i < N; ++i) {
        printf("Does this person have a name or a number?\n");
        printf("Type 1 for name, 0 for number: ");
        scanf("%d", &a[i].which);

        if (a[i].which == 0) {
            printf("Please type the number: ");
            scanf("%d", &a[i].label.number);
        } else {
            printf("Please type the name: ");
            a[i].label.name = mygetline();
        }
    }

    for (i = 0; i < N; ++i) {
        if (a[i].which == 0) {
            printf("%d\n", a[i].label.number);
        } else {
            printf("%s\n", a[i].label.name);
            free(a[i].label.name);
        }
    }

    /* Another way to write the two for loops above. */
    for (p = a; p < a + N; ++p) {
        printf("Does this person have a name or a number?\n");
        printf("Type 1 for name, 0 for number: ");
        scanf("%d", &p->which);

        if (p->which == 0) {
            printf("Please type the number: ");
            scanf("%d", &p->lab.number);
        } else {
            printf("Please type the name: ");
            p->label.name = mygetline();
        }
    }

    for (p = a; p < a + N; ++p) {
        if (p->which == 0) {
            printf("%d\n", p->label.number);
        } else {
            printf("%s\n", p->label.name);
            free(p->label.name);
        }
    }

    return EXIT_SUCCESS;
}
```

I/O using scanf and printf: pp. 151−152

The standard i/o library functions **`scanf`**, **`getchar`**, **`printf`**, and **`putchar`** input from and output to whatever source and destination were stated or implied in the command line. By default, the source is the terminal's keyboard and the destination is the terminal's screen.

A **`<`** in the command line, however, will cause all the **`scanf`**'s and **`getchar`**'s in the program to read their input from a file instead of from the keyboard. And a **`>`** or **`>>`** in the command line will cause all the **`printf`**'s and **`putchar`**'s in the program to write their output to a file instead of to the screen. We used this in Homeworks 5.3 and 5.4.

`1 prog`	*input from keyboard, output to screen*
`2 prog < infile`	*input from* **`infile`**, *output to screen*
`3 prog > outfile`	*input from keyboard, output to* **`outfile`**
`4 prog < infile > outfile`	*input from* **`infile`**, *output to* **`outfile`**
`5 prog1 > tempfile`	
`6 prog2 < tempfile`	
`7 remove the tempfile`	
`8 prog1 | prog2`	**`prog1`** *inputs from keyboard, outputs directly to* **`prog2`**.
	`prog2` *inputs directly from* **`prog1`**, *outputs to screen.*

I/O using the f- and s- functions

Use **`scanf`** to read input from one of three sources specified in the command line: the keyboard, a file, or another program. Use **`fscanf`** to read input from more than one source (e.g., two files, or the keyboard and one file, etc.). **`fscanf`** reads input from a source specified in your program instead of in the command line. You can have several **`fscanf`**'s which each read input from a different source.

Use **`printf`** to write output to one of three destinations specified in the command line: the screen, a file, or another program. Use **`fprintf`** to write output to more than one destination (e.g., two files, or to the screen and one file, etc.). **`fprintf`** writes output to a destination that you can specify in your program instead of in the command line. You can have several **`fprintf`** statements which each write output to a different destination.

Thus there are two reasons to use **`fscanf`** and **`fprintf`** instead of **`scanf`** and **`printf`**: (1) to perform i/o with more than one source or more than one destination; (2) to let the program choose its own source(s) and destination(s) instead of being at the mercy of the command line.

Output to more than one file: pp. 160, 242

`fprintf` takes a "pointer to a **`FILE`**" as its first argument. First call **`fopen`** to get one of these pointers and save the return value in a variable for later use. Tell **`fopen`** that you intend to do one of:

second argument of **`fopen`**	*shell language*
`"r"`	**`<`**
`"w"`	**`>`**
`"a"`	**`>>`**

Page 242 says that you may have at most **`FOPEN_MAX`** files open simultaneously. Many other things can go wrong, too; the variable **`errno`** in K&R p. 248 will tell you which one.

The **`fprintf`** in line 26 outputs exactly six bytes. It does not output the **`'\0'`** at the end of the string.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <errno.h>      /* contains the declaration int errno; */
 4
 5 int main(int argc, char **argv)
 6 {
 7     FILE *out1;
 8     FILE *out2;
 9     const int i = 10;
10
11     /* Open two output files.
12     Clobber them if they already exist; create them if they don't. */
13
14     out1 = fopen("/home1/a/abc1234/outfile1", "w");
15     if (out1 == NULL) {
16         printf("%s: error %d opening outfile1.\n", argv[0], errno);
17         return EXIT_FAILURE;
18     }
19
20     out2 = fopen("/home1/a/abc1234/outfile2", "w");
21     if (out2 == NULL) {
22         printf("%s: error %d opening outfile2.\n", argv[0], errno);
23         return EXIT_FAILURE;
24     }
25
26     fprintf(out1, "hello\n");
27     fprintf(out2, "The answer is %d\n", i);
28
29     if (fclose(out1) != 0) {
30         printf("%s: error %d closing outfile1.\n", argv[0], errno);
31         return EXIT_FAILURE;
32     }
33
34     if (fclose(out2) != 0) {
35         printf("%s: error %d closing outfile2.\n", argv[0], errno);
36         return EXIT_FAILURE;
37     }
38
39     return EXIT_SUCCESS;
40 }
```

Lines 7 and 14 may be combined to

```
 7     FILE *out1 = fopen("/home1/a/abc1234/outfile1", "w");
```

Or lines 14−15 may be combined to

```
14     if ((out1 = fopen("/home1/a/abc1234/outfile1", "w")) == NULL) {
```

just like

```
    while ((c = getchar()) != EOF) {
```

Ditto for lines 20−21.

Perform i/o with any terminal under Unix

Terminals as well as files have names under Unix, so if you know the name of a terminal you can `fopen` it like a file:

```
    FILE *out = fopen("/dev/pts/100", "w");
    fprintf(out, "hello\n");
    fclose(out);
```

Input from more than one file

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>           /* contains the declaration int errno; */

int main(int argc, char **argv)
{
    FILE *in1;
    FILE *in2;
    int i;
    int j;

    /* Open two input files. */

    in1 = fopen("/home1/a/abc1234/infile1", "r");
    if (in1 == NULL) {
        printf("%s: error %d opening infile1.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    in2 = fopen("/home1/a/abc1234/infile2", "r");
    if (in2 == NULL) {
        printf("%s: error %d opening infile2.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    fscanf(in1, "%d", &i);
    fscanf(in2, "%d", &j);

    if (fclose(in1) != 0) {
        printf("%s: error %d closing infile1.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    if (fclose(in2) != 0) {
        printf("%s: error %d closing infile2.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

What is a “pointer to a FILE”?

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main(int arc, char **argv)
 5 {
 6     FILE *in, *out;
 7     int i;
 8
 9     in = fopen("/home1/a/abc1234/infile", "r");
10     if (in == NULL) {
11         fprintf(stderr, "%s: couldn’t open input file\n", argv[0]);
12         return EXIT_FAILURE;
13     }
14
15     out = fopen("/home1/a/abc1234/outfile", "w");
16     if (out == NULL) {
17         fprintf(stderr, "%s: couldn’t open output file\n", argv[0]);
18         return EXIT_FAILURE;
19     }
20
21     scanf("%d", &i);                                          /* _iob[0] */
22     printf("hello\n");                                        /* _iob[1] */
23     fprintf(stderr, "%s: Permission denied\n", argv[0]);      /* _iob[2] */
24     fscanf(in, "%d", &i);                                     /* _iob[3] */
25     fprintf(out, "hello\n");                                  /* _iob[4] */
```

Here is the contents of the **`_iob`** array immediately after line 25:

	buffer	**_cnt**	**_bufsiz**	**_flag**	**_file**
_iob[0]	\n	1	8192	00000 01001	0
_iob[1]		0	8192	00100 01010	1
_iob[2]		0	0	00000 00110	2
_iob[3]	\n	1	8192	00000 01001	3
_iob[4]	hello\n	8186	8192	00000 01010	4

```
_____ _____
IIIII IIIII
OOOOO OOOOO
ARLSE EMNWR
PWBTR OYBRE
P FRR FBFTA
E  G   U  D
N      F

D
```

Each call to **`fprintf`** does not necessarily send its output to the outside world immediately. Instead, the data may be stored temporarily in region of memory called a *buffer*. It may take many calls to `fprintf` before the buffer is full. When this happens the buffer is *flushed:* all the data in the buffer is output at once, and the buffer is emptied.

The buffer has several attendant variables: `_cnt`, `_bufsize`, `_flag`, `_file`, all stored in a structure of data type `struct _iobuf` declared in `<stdio.h>`. Each output file (and each input file) has its own buffer and variables, so there is an array of these structures named `_iob` in the file `data.c` of the standard i/o library.

`FILE` is another name for the `struct _iobuf` data type. Thus a "pointer to a `FILE`" is a variable that can hold the address of an element of the `_iob` array, which is why the linker had to add this file to your program in Handout 8, p. 18. `_iob` has one element for each file with which your program can perform i/o. The function `fopen` adds a new element to `_iob` and returns the address of the new element. The function `fclose` removes an element from `_iob`.

▼ Homework 9.1: required reading: p. 176

Look at the `_iob` array of structures declared in the file `<stdio.h>` (i.e., `/usr/include/stdio.h` in Unix or `Z:\BIN\BC3\INCLUDE\STDIO.H` in Borland C++ under DOS). When we do i/o with multiple files, there is one structure in this array for each open file. Thus the maximum number of files that can be open at once (`FOPEN_MAX` in p. 242) is merely the maximum number of structures in this array.

```
/* Excerpts from the file /usr/include/stdio.h */

typedef unsigned int size_t;     /* type of sizeof */

#define EOF (-1)           /* returned by getchar on end of file */
#define NULL 0             /* returned by malloc and realloc on failure */
#define FOPEN_MAX 64
#define FILENAME_MAX 1024  /* longest filename, p. 242 */
#define BUFSIZ 1024        /* size in bytes of each i/o buffer */

extern struct _iobuf {
    int _cnt;              /* number of characters left in the buffer */
    char *_ptr;            /* address of the next character in the buffer */
    char *_base;           /* address of the start of the buffer */
    int _bufsiz;           /* the size of the buffer, in bytes */
    short _flag;           /* individual bits shown below */
    short _file;           /* file descriptor passed to read & write, K&R p. 170 */
} _iob[_N_STATIC_IOBS];

typedef struct _iobuf FILE;

#define stdin     (&_iob[0])
#define stdout    (&_iob[1])
#define stderr    (&_iob[2])

/* Individual bits within the _flag field of struct _iobuf. */
#define _IOREAD   00001    /* This is an input file. */
#define _IOWRT    00002    /* This is an output file. */
#define _IONBF    00004    /* No buffering: do all i/o immediately. */
#define _IOMYBUF  00010    /* Using buffer not supplied by setvbuf. */
#define _IOEOF    00020    /* Has reached the end of file (only for input files). */
#define _IOERR    00040    /* Has encountered an error. */
#define _IOSTRG   00100
#define _IOLBF    00200    /* Line buffering: flush at every newline. */
#define _IORW     00400    /* This is both an input and an output file. */
```

```

/* Third argument of the fseek function and lseek system call. */
#define SEEK_SET 0         /* count forward from start of file */
#define SEEK_CUR 1         /* count forward from current location */
#define SEEK_END 2         /* count forward from end of file */
```

▲

Three aboriginal entries

The first three elements in the **`_iob`** array are automatically created when your C program starts running. The three special "pointers to **`FILE`**" **`stdin`**, **`stdout`**, **`stderr`** are the addresses of these three elements. They input from and output to whatever source and destination were stated or implied in the command line:

```
    int i;
    int c;

    fprintf(&_iob[1], "hello\n");       putc(c, &_iob[1]);
    fprintf(stdout, "hello\n");         putc(c, stdout);
    printf("hello\n");                  putchar(c);

    fscanf(&_iob[0], "%d", &i);         c = getc(&_iob[0]);
    fscanf(stdin, "%d", &i);            c = getc(stdin);
    scanf("%d", &i);                    c = getchar(c);

    fprintf(&_iob[2], "%s: Permission denied\n", argv[0]);
    fprintf(stderr, "%s: Permission denied\n", argv[0]);
```

Send an error message to the stderr

Always write the error message on **`stderr`** with **`fprintf`**, rather than on **`stdout`** with **`printf`**. You don't need to open or close **`stderr`**: it's done for you automatically. Begin the error message with the name of the program, a colon, and a blank. Then return **`EXIT_FAILURE`** to tell the program which ran this program that something failed.

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

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "%s: requires one command line argument.\n", argv[0]);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

I/O one variable at a time: K&R pp. 153−159; King pp. 486−498

	standard i/o	*file*	*memory*
input	**`scanf`**	**`fscanf`**	**`sscanf`**
output	**`printf`**	**`fprintf`**	**`sprintf`**

I/O one character at a time: K&R p. 161; King pp. 498−501

	standard i/o	*file*
input	`getchar()`	`getc(fp)`
output	`putchar(c)`	`putc(c, fp)`

I/O one line at a time: K&R pp. 164−165; King pp. 501−502

	standard i/o	*file*
input	`gets(p)`	`fgets(p, n, fp)`
output	`puts(p)`	`fputs(p, fp)`

sprintf and sscanf

`sprintf` is just like **`printf`**, except that the characters are deposited in memory instead of being displayed on the screen. The first argument of **`sprintf`** specifies the memory address where the characters should go. See Handout 6, p. 2, the second line 13. The following example puts the number 49 (i.e., the ASCII code of the character **`'1'`**) into **`a[0]`**, the number 50 (i.e., the ASCII code of the character **`'2'`**) into **`a[1]`**, etc., and a terminating **`'\0'`** into **`a[5]`**.

```
    int i = 12345;
    char a[256];

    sprintf(a, "%d", i);
    printf("%s\n", a);
```

```
12345
```

`sscanf` is just like **`scanf`**, except that the input characters are taken from memory instead of from the keyboard. The first argument of **`sscanf`** specifies the memory address from which the characters should be taken. See Handout 8, p. 4, line 31 for an example of **`sscanf`**. The following example puts the number 12,345 into **`i`**.

```
    int i;
    char a[] = "12345";

    sscanf(a, "%d", &i);
    printf("%d\n", i);
```

The first argument of **`sprintf`** and **`sscanf`** does not necessarily have to be the starting address of an array of **`char`**'s. It can be any memory address at all, e.g., the address of a block of memory allocated by **`malloc`**.

gets and puts: K&R p. 247; King pp. 246−249

`gets(line)` is an easier way to do

```
    scanf("%[^\n]", line);       /* Handout 4, p. 9, line 16 */
```

The program on K&R p. 17 copies input to output one **`char`** at a time.

```
/* Copy the input to the output, one line at a time. */
#include <stdio.h>
#include <stdlib.h>
#define N   256     /* maximum line length */

int main()
```

```
{
    char line[N];

    while (gets(line) != NULL) {
        puts(line);
    }

    return EXIT_SUCCESS;
}
```

Treat a file as a one-dimensional array of characters: K&R p. 248; King pp. 504−507

To write the characters **"abc"** into positions 100, 101, and 102 in a file, call **fseek** before you **fprintf**. The position numbers start at 0 and are measured in bytes. Use a **long** variable to hold these numbers. **SEEK_SET** means to count the distance (in this case, 100 bytes) from the *start* of the file.

Calling **fseek** before **fprintf** will let you write into any byte of the file in any order you choose, i.e., it gives you random access. You can also call **fseek** before **fscanf** to read input from any position in the file.

```
/* Excerpt from /usr/include/stdio.h: third argument of fseek */
#define SEEK_SET 0   /* count from start of file */
#define SEEK_CUR 1   /* count from current position */
#define SEEK_END 2   /* count from end of file */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    FILE *out;
    long position;

    out = fopen("/home1/a/abc1234/arrayfile", "w");
    if (out == NULL) {
        fprintf(stderr, "%s: error %d opening output file.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    fseek(out, 100, SEEK_SET);      /* Return 0 if successful. */
    position = ftell(out);          /* Put 100 into position. */

    fprintf(out, "%s", "abc");
    position = ftell(out);          /* Put 103 into position. */

    fclose(out);
    return EXIT_SUCCESS;
}
```

How not to treat a file as a one-dimensional array of int's

Now that we can randomly access any byte in a file with **fseek**, let's treat the file as a one-dimensional array of **int**'s.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <errno.h>

int a[] = {
    1,
    1234,
    12345,
};
#define N   (sizeof a / sizeof a[0])

int main(int argc, char *argv[])
{
    int i;
    FILE *out;

    out = fopen("/home1/a/abc1234/arrayfile", "w");
    if (out == NULL) {
        fprintf(stderr, "%s: error %d opening arrayfile.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    for (i = 0; i < N; ++i) {
        fprintf(out, "%d\n", a[i]);
    }

    fclose(out);
    return EXIT_SUCCESS;
}
```

The file **`arrayfile`** contains 13 bytes: ten digits and three newlines.

```
1
1234
12345
```

Each number occupies a different amount of space in the file: 2 bytes, 5 bytes, and 6 bytes respectively. Therefore the first number begins at position 0, the second at position 2, and the third at position 7.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    FILE *in;
    int i;

    in = fopen("/home1/a/abc1234/arrayfile", "r");
    if (in == NULL) {
        fprintf(stderr, "%s: error %d opening arrayfile.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    fseek(in, 0, SEEK_SET);  /* Travel to 1st number. */
    fscanf(in, "%d", &i);

    fseek(in, 7, SEEK_SET);  /* Travel to 3rd number. */
    fscanf(in, "%d", &i);

    fseek(in, 2, SEEK_SET);  /* Travel to 2st number. */
    fscanf(in, "%d", &i);

    fclose(in);
    return EXIT_SUCCESS;
}
```

A simple solution that wastes space

To treat the file as an array of **`int`**'s, each **`int`** must be stored in the same amount of space. We could allow 6, 11, or 20 **`char`**'s for the longest possible string of characters (**`-32768`** if **`sizeof(int) == 2`**, **`-2147483648`** if **`sizeof(int) == 4`**, **`-9223372036854775808`** if **`sizeof(int) == 8`**): but this would waste space in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define FORMAT "%6d\n"       /* big enough to hold "-32768" */

int a[] = {
    1,
    1234,
    12345,
};
#define N   (sizeof a / sizeof a[0])

int main(int argc, char *argv[])
{
    int i;
    FILE *out;

    out = fopen("/home1/a/abc1234/arrayfile", "w");
    if (out == NULL) {
        fprintf(stderr, "%s: error %d opening arrayfile.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    for (i = 0; i < N; ++i) {
        fprintf(out, FORMAT, a[i]);
    }

    fclose(out);
    return EXIT_SUCCESS;
}
```

Assuming **`sizeof(int) == 2`**, the file **`arrayfile`** contains 21 bytes: ten digits, eight blanks, and three newlines.

```
     1
  1234
 12345
```

Now each number occupies the same amount of space in the file: seven bytes. Therefore the first number begins at position 0, the second at position 7, and the third at position 14.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define N 7  /* number of bytes occupied by each number in the file */

int main()
{
    FILE *in;
    int i;

    in = fopen("/home1/a/abc1234/arrayfile", "r");
    if (in == NULL) {
        fprintf(stderr, "%s: error %d opening arrayfile.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    fseek(in, 0 * N, SEEK_SET);  /* Travel to 1st number. */
    fscanf(in, "%d", &i);

    fseek(in, 2 * N, SEEK_SET);  /* Travel to 3rd number. */
    fscanf(in, "%d", &i);

    fseek(in, 1 * N, SEEK_SET);  /* Travel to 2st number. */
    fscanf(in, "%d", &i);

    fclose(in);
    return EXIT_SUCCESS;
}
```

Treat a file as a one-dimensional array of int’s (or any other data type): K&R p. 247; King p. 502

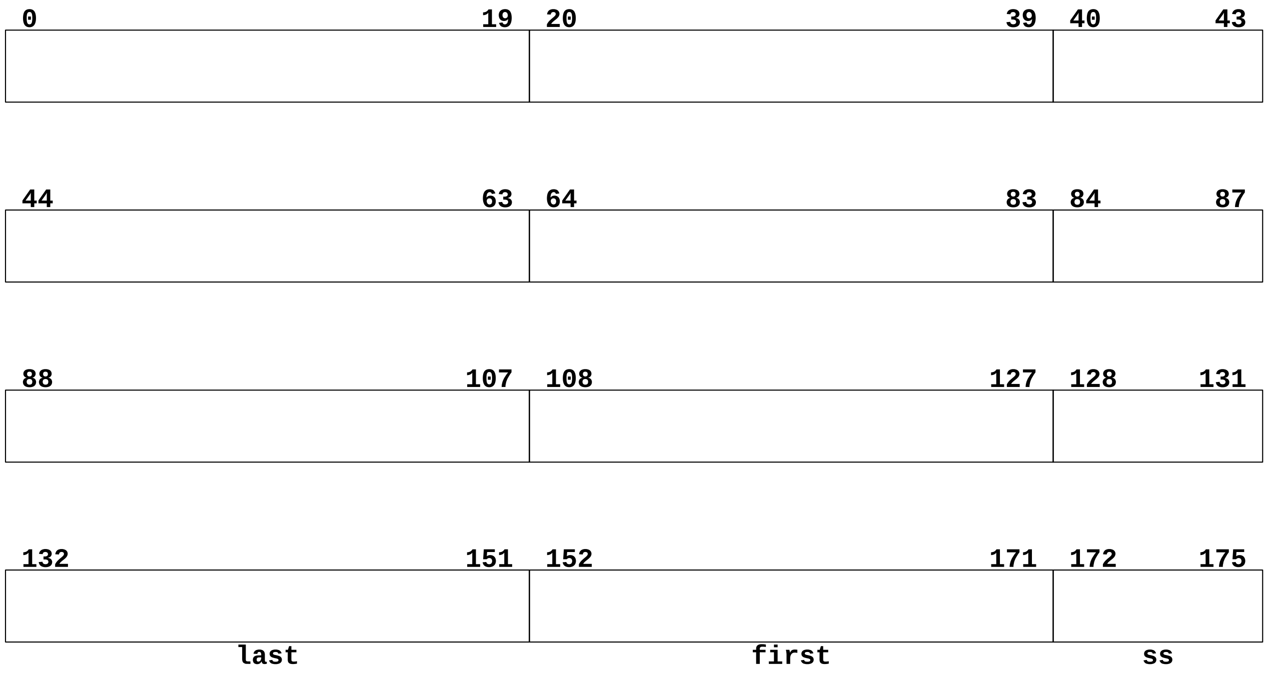
Instead of writing an **`int`** as a string of ASCII characters with an optional minus sign, we will write it as **`sizeof(int)`** binary bytes. Since every **`int`** in the file now occupies the same number of bytes (viz., **`sizeof(int)`**), we can now **`fseek`** directly to **`int`** number *n*. The price you pay for this data compression is that **`arrayfile`** no longer consists of humanly readable ASCII characters. Read it with **`fread`** instead of with **`fscanf`**. (On some operating systems, you may have to **`fopen`** the file with **`"wb"`** instead of **`"w"`**, K&R pp. 160, 242; King pp. 479−480.)

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int a[] = {
    1,
    1234,
    12345,
};
#define N   (sizeof a / sizeof a[0])
```

```

int main(int argc, char *argv[])
{
    int i;
    FILE *out;

    out = fopen("/home1/a/abc1234/arrayfile", "w");
    if (out == NULL) {
        fprintf(stderr, "%s: error %d opening arrayfile.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    for (i = 0; i < N; ++i) {
        fwrite(a + i, sizeof a[i], 1, out);
    }

    fclose(out);
    return EXIT_SUCCESS;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    FILE *in;
    int i;

    in = fopen("/home1/a/abc1234/arrayfile", "r");
    if (in == NULL) {
        fprintf(stderr, "%s: error %d opening arrayfile.\n", argv[0], errno);
        return EXIT_FAILURE;
    }

    fseek(in, 0 * sizeof i, SEEK_SET);   /* Travel to 1st number. */
    fread(&i, sizeof i, 1, in);

    fseek(in, 2 * sizeof i, SEEK_SET);   /* Travel to 3rd number. */
    fread(&i, sizeof i, 1, in);

    fseek(in, 1 * sizeof i, SEEK_SET);   /* Travel to 2st number. */
    fread(&i, sizeof i, 1, in);

    fclose(in);
    return EXIT_SUCCESS;
}
```

Treat a file as a one-dimensional array of structures



```c
/* Excerpts from stddef.h. */
typedef unsigned int size_t;   /* result type of the sizeof operator */
#define offsetof(s_name, m_name) ((size_t)&((s_name *)0)->m_name)
```

```c
/* This file is business.h, describing the layout of the records that constitute
a file called FILENAME.  The two names within each record can each hold up to
NAMELEN-1 characters.  A ’\0’ follows the last character of each name. */

#define FILENAME   "/home1/a/abc1234/business.data"
#define NAMELEN 20

typedef struct {
    char last[NAMELEN];      /* last name */
    char first[NAMELEN];     /* first name */
    long ss;                 /* social security number */
} record_t;

typedef long recno_t;            /* record number, starting at 0 */
#define MAXRECORDS 10        /* the number of records in the file */
```

```c
/* Create the file and let the user fill it up with records. */
#include <stdio.h>
#include <stdlib.h>
#include "business.h"

int main(int argc, char **argv)
{
    FILE *fp;
    record_t record;
    recno_t r;

    fp = fopen(FILENAME, "w");
    if (fp == NULL) {
        fprintf(stderr, "%s: can’t open file %s\n", argv[0], FILENAME);
        return EXIT_FAILURE;
    }

    printf("Please type last name, first name, and ss.\n");
    printf("Press RETURN after each of these three items.\n");

    for (r = 0; r < MAXRECORDS; ++r) {
        printf("Record number %ld:\n", r);
        scanf("%s%s%ld", record.last, record.first, &record.ss);
        fwrite(&record, sizeof record, 1, fp);
    }

    fclose(fp);
    return EXIT_SUCCESS;
}
```

```
/* Let the user change one field of one record. */
#include <stdio.h>
#include <stdlib.h>
#include "business.h"

int main(int argc, char **argv)
{
    FILE *fp;
    recno_t r;
    int field;
    long offset;     /* offset in bytes from start of file */
    record_t record;

    fp = fopen(FILENAME, "r+");      /* input & output, K&R p. 242; King pp. 479−480 */
    if (fp == NULL) {
        fprintf(stderr, "%s: can’t open file %s\n", argv[0], FILENAME);
        retrn EXIT_FAILURE;
    }

    printf("Which record number would you like to update? ");
    scanf("%ld", &r);
    printf("Which field within record number %ld would you like to update?\n", r);

    printf("0  last name\n");
    printf("1  first name\n");
    printf("2  social security number\n");
    scanf("%d", &field);

    if (field == 2) {
        /* The total offset is the distance in bytes from the start of
        the file to the start of the record we’re interested in, plus
        the distance in bytes from the start of the record to the start
        of the field we’re interested in.  See Handout 3, p. 13, (6). */
        offset = r * sizeof(record_t) + offsetof(record_t, ss);
35
36          fseek(fp, offset, SEEK_SET);
37          fread(&record.ss, sizeof record.ss, 1, fp);
38
39          printf("The current value of this field is %ld.\n", record.ss);
40          printf("Please type the value with which to replace it: ");
41          scanf("%ld", &record.ss);
42
43          fseek(fp, offset, SEEK_SET);
44          fwrite(&record.ss, sizeof record.ss, 1, fp);
45      }
46
47      fclose(fp);
48      return EXIT_SUCCESS;
49  }
```

▼ **Homework 9.2: complete the above program**

Allow the user to update the `last` or `first` field in addition to the `ss` field:

```
    if (field == 0) {
        /* write code here to update the last name */
    } else if (field == 1) {
        /* write code here to update the first name */
    } else if (field == 2) {
        /* lines 29-43 shown above go here */
    } else {
        /* error message */
    }
```

▲

▼ **Homework 9.3: change the if's to an array**

✎ Extra credit. Instead of the `if`'s in the previous homework, make an array of structures:

```
#include <stddef.h>

typedef struct {
    char *name;            /* name of this field */
    char *printf_format;   /* display current value on screen */
    char *scanf_format;    /* let user input new value */
    size_t size;           /* sizeof this field in bytes */
    size_t offset;         /* distance in bytes from start of record to this field */
} field_t;

/* Must declare record before you can use its address in array initialization */
record_t record;

field_t a[] = {
    {"last name",  "%s",   "%s",   sizeof record.last,  offsetof(record_t, last)},
    {"first name", "%s",   "%s",   sizeof record.first, offsetof(record_t, first)},
    {"ss number",  "%ld",  "%ld",  sizeof record.ss,    offsetof(record_t, ss)},
};
#define NFIELDS (sizeof a / sizeof a[0])
```

▲

A pointer to a function: K&R pp. 118−121; King pp. 385−391

The second pair of parentheses in line 13 is the function call operator (line 1 in the table on K&R p. 53; King p. 595). The first pair of parentheses in line 13 are necessary to force the unary `*` operator (line 2 in the table on K&R p. 53; King p. 595) to execute before the function call operator.

In newer versions of C (and in C++), you don't need the `*` (and therefore you also don't need the parentheses around the `*p`) in lines 13 and 16. See the fourth from last bullet on K&R p. 260; King p. 386.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>
 4
 5 int main()
 6 {
 7     double d;
 8     double (*p)(double) = sqrt;     /* Put the address of the function sqrt into p. */
 9
10     printf("%p\n", p);              /* Find out where sqrt is. */
11     printf("%p\n", sqrt);           /* Another way to do the same thing. */
12
13     d = (*p)(2.0);                  /* Call the function sqrt, put return val into d. *
14     printf("%f\n", d);
15
16     printf("%f\n", (*p)(2.0));
17     return EXIT_SUCCESS;
18 }
```

You can split line 8 into

```
 8     double (*p)(double);
 9     p = sqrt;
```

—but why would you want to?

Use an array of pointers to functions to avoid a chain of if's

Here is a function whose only job is to call one of five other functions depending on the argument it receives. See K&R pp. 118−121 for a more complicated use of pointers to functions.

```
void f0(void);
void f1(void);
void f2(void);
void f3(void);
void f4(void);

void error(void);                  /* print a message and exit */

void f(int n);
{
    if (n == 0) {
        f0();
    } else if (n == 1) {
        f1();
    } else if (n == 2) {
        f2();
    } else if (n == 3) {
        f3();
    } else if (n == 4) {
        f4();
    } else {
        error();
    }
}
```

```
void f0(void);
void f1(void);
void f2(void);
void f3(void);
void f4(void);

void error(void);                    /* print a message and exit */

void (*a[])(void) = {                /* array of pointers to functions */
    f0,                              /* the address of the function f0 */
    f1,                              /* the address of the function f1 */
    f2,                              /* the address of the function f2 */
    f3,                              /* the address of the function f3 */
    f4                               /* the address of the function f4 */
};
#define N (sizeof a / sizeof a[0])  /* number of functions to choose from */

void f(int n)
{
    if (0 <= n && n < N) {
        (*a[n])();
    } else {
        error();
    }
}
```

An array of structures, each of which contains a pointer to a function

To make this a complete program, you must define the functions declared in lines 4−7.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void save(void);
 5 void save_as(void);
 6 void print(void);
 7 void quit(void);
 8
 9 typedef struct {
10     int n;
11     char *message;
12     void (*f)(void);    /* pointer to a function */
13 } selection_t;
14
15 selection_t a[] = {
16     {1,     "To save your file",                         save},
17     {2,     "To save your file with a different name",   save_as},
18     {3,     "To print your file",                        print},
19     {9,     "To quit",                                   quit},
    {-1,    NULL,                                       NULL}
};

int main(int argc, char **argv)
{
    int i;
    selection_t *p;

    printf("Indicate your choice and press RETURN.\n");
    for (p = a; p->message != NULL; ++p) {
        printf("%s, type %d.\n", p->message, p->n);
    }

    scanf("%d", &i);

    for (p = a; p->message != NULL; ++p) {
        if (p->n == i) {
            (*p->f)();   /* call the function whose address is in the structure */
            return EXIT_SUCCESS;
        }
    }

    fprintf(stderr, "%s: invalid option %d\n", argv[0], i);
    return EXIT_FAILURE;
}
```

Pass a pointer to one function to another function

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/funcptr1.c`

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

int hello(int n);
int goodbye(int n);
void f(int (*p)(int));

int main()
{
    f(hello);
    f(goodbye);
    return EXIT_SUCCESS;
}

void f(int (*p)(int))
{
    printf("%d\n", (*p)(10));
}

int hello(int n)
{
    printf("hello\n");
    return 2 * n;
}
```

```

int goodbye(int n)
{
    printf("goodbye\n");
    return 3 * n;
}
```

```
hello
20
goodbye
30
```

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/funcptr2.c

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

int hello(int n);
int goodbye(int n);

/* An fp_t is a pointer to a function that has one int argument and that
returns an int.  For example, an fp_t could hold the address of the
function hello or the function goodbye. */

typedef int (*fp_t)(int);

void f(fp_t p);

int main()
{
    f(hello);
    f(goodbye);
    return EXIT_SUCCESS;
}

void f(fp_t p)
{
    printf("%d\n", (*p)(10));
}

int hello(int n)
{
    printf("hello\n");
    return 2 * n;
}

int goodbye(int n)
{
    printf("goodbye\n");
    return 3 * n;
}
```

qsort in the C Standard Library

See Stroustrup pp. 158, 334.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/sort/qsort.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>   //for qsort
 3
 4 int cmp(const void *p1, const void *p2);
 5
 6 int main()
 7 {
 8     int a[] = {1, 3, 0, 2, 5};
 9     const size_t n = sizeof a / sizeof a[0];
10     const int *p;
11
12     qsort(a, n, sizeof a[0], cmp);   //last argument is a pointer
13
14     for (p = a; p < a + n; ++p) {
15         printf("%d\n", *p);
16     }
17
18     return EXIT_SUCCESS;
19 }
20
21 int cmp(const void *p1, const void *p2)
22 {
23     const int i = *(const int *)p1;
24     const int j = *(const int *)p2;
25
26     if (i < j) {
27         return -1;
28     }
29
30     if (i > j) {
31         return 1;
32     }
33
34     return 0;
35 }
```

The above lines 26−32 may be combined to

```
36     return i < j ? -1 : i > j;
```

□