Fall 2006 Handout 11

The comma operator: K&R pp. 62–63; King pp. 94–95

The comma operator lets you squeeze two expressions into a place where the syntax permits only one. You will use it only in a **for** loop :

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6     char *name[] = {
 7         "Mercury",
 8         "Venus",
 9         "Earth",
10         "Mars",
11         "Jupiter",
12         "Saturn",
13         "Uranus",
14         "Neptune",
15         "Pluto"
16     };
17 #define N (sizeof name / sizeof name[0])
18
19     double factor[] = {
20          .27,   /* Mercury */
21          .85,   /* Venus */
22         1.00,   /* Earth */
23          .38,   /* Mars */
24         2.33,   /* Jupiter */
25          .92,   /* Saturn */
26          .85,   /* Uranus */
27         1.12,   /* Neptune */
28          .44,   /* Pluto */
29     };
30
31     char **pname;
32     double *pfactor;
33
34     for (pname = name, pfactor = factor; pname < name + N; ++pname, ++pfactor) {
35         printf("%s %f\n", *pname, *pfactor);
36     }
37
38     return EXIT_SUCCESS;
39 }
```

Of course, the above example can be done with only one pointer if we change the two parallel arrays into one array of structures.

 All rights reserved ©2006 Mark Meretzky

enum: K&R p. 39, 259; King pp. 350–353

```
 1 /* The b field has one of the following values:
 2     0 Gold
 3     1 Juneau
 4     2 Omaha
 5     3 Sword
 6     4 Utah
 7 */
 8
 9 typedef struct {
10     char *name;
11     int b;              /* which beach they landed on */
12 } unit_t;
13
14 unit_t a[] = {
15     {"99th Airborne",         2},
16     {"101st Airborne",        4},
17     {"99th Amphibious",       0},
18     {"347th Antiaircraft",    0},
19     {NULL,                    -1}
20 };

21 /* The b field has one of the following values: */
22 #define GOLD     0
23 #define JUNEAU   1
24 #define OMAHA    2
25 #define SWORD    3
26 #define UTAH     4
27
28 typedef struct {
29     char *name;
30     int b;              /* which beach they landed on */
31 } unit_t;
32
33 unit_t a[] = {
34     {"99th Airborne",         OMAHA},
35     {"101st Airborne",        UTAH},
36     {"99th Amphibious",       GOLD},
37     {"347th Antiaircraft",    GOLD},
38     {NULL,                    -1}
39 };

40 #include <stdio.h>
41 #include <stdlib.h>
42
43 typedef enum {
44     gold,
45     juneau,
46     omaha,
47     sword,
48     utah
49 } beach_t;
50
51 typedef struct {
```

 All rights reserved ©2006 Mark Meretzky

```
52     char *name;
53     beach_t b;
54 } unit_t;
55
56 unit_t a[] = {
57     {"99th Airborne",         omaha},
58     {"101st Airborne",        utah},
59     {"99th Amphibious",       gold},
60     {"347th Antiaircraft",    gold},
61     {NULL,                    -1}
62 };
63
64 int main()
65 {
66     unit_t *p;
67
68     printf("The following units landed at Gold:\n");
69
70     for (p = a; p->name != NULL; ++p) {
71         if (p->b == gold) {
72             printf("%s\n", p->name);
73         }
74     }
75
76     return EXIT_SUCCESS;
77 }
```

Lines 43–49 above create a new data type:

```
 1     int i = 10;          /* one of 65,536 or 4,294,967,296 possible values */
 2     char c = 'A';        /* one of 256 possible values */
 3     beach_t b = omaha;   /* one of 5 possible values */
```

Another example:

```
 1 typedef enum {
 2     jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov dec
 3 } month_t;
 4
 5 typedef struct {
 6     month_t m;
 7     int d;
 8     int y;
 9 } date_t;
10
11 date_t a[] = {
12     {jan,  1, 2006},
13     {feb, 28, 2006},
14     {jul,  4, 2006},
15     {dec, 25, 2006},
16     {dec, 31, 2006},
17 };
```

 All rights reserved ©2006 Mark Meretzky

Start the enums off at whatever value you want

```
1 typedef enum {
2     land = 1,
3     sea
4 } by_t;
```

Things that happen automatically in other languages, but not in C

When you ask for something in C, only the bare minimum amount of work is performed. It does nothing beyond what you ask for explicitly. When you create a variable, array, or structure in C, it creates somthing that is just big enough to hold the data you describe. It allocates no space beyond what you ask for explicitly.

(1) When a variable is created as the program runs, it is given no initial value.

(2) When you assign a value to a variable, the computer does not check that the variable is big enough to hold the value.

(3) When you divide, the computer does not first check if the divisor is zero.

(4) When you create an array of 10 **int**'s:

```
int a[10];
```

the total number of bytes occupied by the array is exactly 10 times **sizeof(int)**. The array contains no additional information such as the number of dimensions or the upper and lower bounds of each dimension.

(5) When you store a value into an array element, the computer does not first check if the subscript is legal.

(6) When you store a value into memory using a pointer,

```
*p = 10;
```

the computer does not first check if the pointer is pointing to a place in which you're allowed to store the value.

(7) When you call a function,

```
f(x, y, z)
```

no information is passed to the function other than the values you write within the parentheses. In other languages, additional information is passed automatically. For example, if the function can take a variable number of arguments or arguments of different data types (e.g., **printf** or **scanf**), other languages will automatically pass the number and data types of the arguments. In C, you have to pass this information explicitly (e.g., in the first argument of **printf** or **scanf**).

(8) When you pass an array to a function, only the smallest possible amount of data is actually moved inside the machine: just the address of the array. In other languages, the entire array is moved down to the function.

(9) If you pass an array to a function and you want the function to know the size of the array, you must pass this information explicitly. In other languages, this information is passed automatically.

(10) No function is ever called unless you call it explicitly. In other languages, many functions (such as the equivalent of **malloc**) are called implicitly:

```
int i;
scanf("%d", &i);
int a[i];            /* legal in other languages, but not in C */
```

 All rights reserved ©2006 Mark Meretzky

Most of the work is done within the expressions

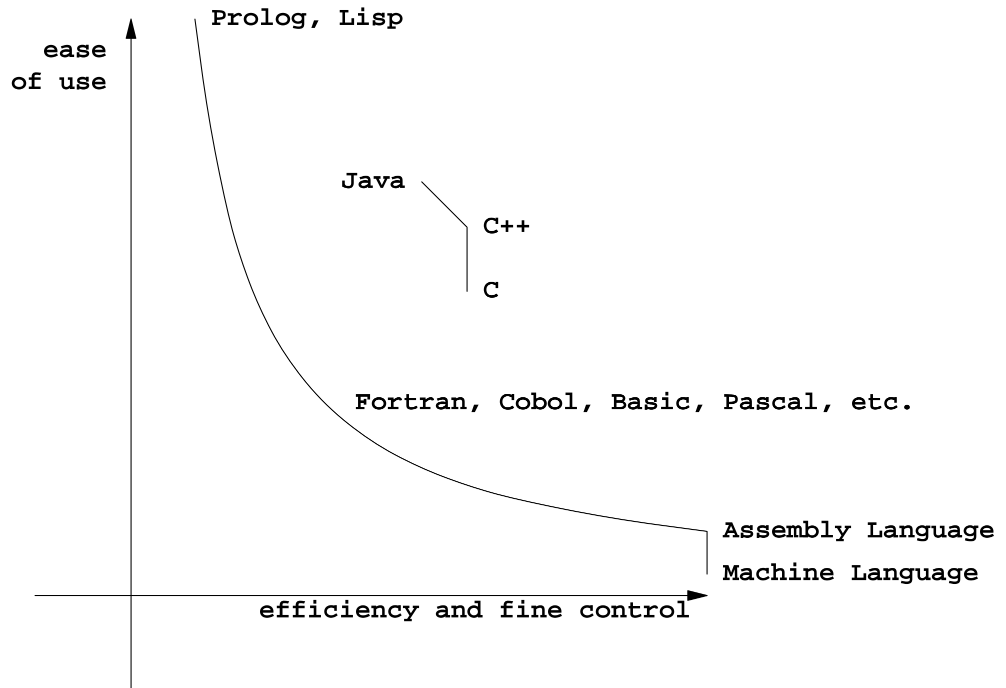
```
 1 a = b == c ? d : e++;              /* Example 1 */
 2
 3 if (b == c) {
 4     a = d;
 5 } else {
 6     a = e;
 7     e = e + 1;
 8 }
```

```
 1 *s++ = *t++;                       /* Example 2 */
 2
 3 *s = *t;
 4 s = s + 1;
 5 t = t + 1;
```

```
 1 while (*s++ = *t++) {              /* Example 3: K&R p. 106 */
 2 }
 3
 4 *s = *t;
 5 s = s + 1;
 6 t = t + 1;
 7 while (s[-1]) {
 8     *s = *t;
 9     s = s + 1;
10     t = t + 1;
11 }
```

 All rights reserved ©2006 Mark Meretzky

The main sequence



Bibliography

☞ Kernighan and Ritchie's *The C Programming Language, Second Edition* is the only C book you need.

☞ In C, we can crudely group variables and functions together by writing them in the same **.c** file as in Homework 8.4: **push**, **pop**, **val**, and **sp**. A more sophisticated way to do this is the **class** feature of C++. See *The C++ Programming Language, Third Edition* by Bjarne Stroustrup; Addison Wesley, 1997; ISBN 0-201-88954-4.

☞ Unix exists to help you write, compile, debug, test, document, copy, and ship big C programs. The most meaty Unix book for beginners is *The Unix Programming Environment* by Brian W. Kernighan and Rob Pike; Prentice-Hall, 1984; ISBN 0-13-937681-X. Take my Unix course (X52.9545) (212) 998-7190.

☞ The most important rules for good programming are the same in all languages. Although the examples are in Fortran and PL/I, the best book about programming is *The Elements of Programming Style, Second Edition* by Brian W. Kernighan and P. J. Plauger; McGraw-Hill, 1978; ISBN 0-07-034207-5.
□

 All rights reserved ©2006 Mark Meretzky