Fall 2006 Handout 8

Pass an array of strings

When we pass any type of array to a function, we’re actually passing the address of the first element of the array. But each element of the array of Chinese years (Handout 6, pp. 4−5) contains the address of the first **`char`** of a string. Therefore when we pass this array to a function, we’re passing the address of an address of a **`char`**, i.e., a “pointer to pointer to **`char`**”.

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

void f(int n, char *a[]);
void g(int n, char **p);

int main()
{
    char *animal[] = {
        "monkey",
        "rooster",
        "dog",
        "pig",
        "rat",
        "ox",
        "tiger",
        "hare",
        "dragon",
        "snake",
        "horse",
        "sheep"
    };
#define N   (sizeof animal / sizeof animal[0])

    f(N, animal);
    g(N, animal);

    return EXIT_SUCCESS;
}

/* Print an array of strings, one per line. */

void f(int n, char *a[])      /* empty [] as in Handout 7, p. 20, line 17 */
{
    int i;

    for (i = 0; i < n; ++i) {
        printf("%s\n", a[i]);
    }
}
```

```

/* Print an array of strings, one per line. */

void g(int n, char **p)
{
    int i;

    for (i = 0; i < n; ++i) {
        printf("%s\n", p[i]);
    }
}
```

Excerpt from stdio.h: K&R p. 102

```
#define NULL 0
```

Another way to indicate where an array ends

Instead of making a

```
#define N   (sizeof animal / sizeof animal[0])
```

to determine the number of elements in an array, we can mark the end of the array with a dummy element. This lets us dispense with the second argument of `f` and `g`. Use `NULL` as the dummy element if the data type of the elements is any kind of pointer.

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

void f(char *a[]);
void g(char **p);

int main()
{
    char *animal[] = {
        "monkey",
        "rooster",
        "dog",
        "pig",
        "rat",
        "ox",
        "tiger",
        "hare",
        "dragon",
        "snake",
        "horse",
        "sheep",
        NULL
    };

    f(animal);
    g(animal);

    return EXIT_SUCCESS;
}

```

```
/* Print an array of strings, one per line. */

void f(char *a[])
{
    int i;

    for (i = 0; a[i] != NULL; ++i) {
        printf("%s\n", a[i]);
    }
}

/* Print an array of strings, one per line. */

void g(char **p)
{
    int i;

    for (i = 0; p[i] != NULL; ++i) {
        printf("%s\n", p[i]);
    }
}
```

Command line arguments: K&R pp. 114−118; Ki pp. 263−264

We emphasized how to pass an array of strings to a function, because the command line arguments of a C program are passed to the `main` function as an array of strings called `argv`.

```
1$ prog hello 10 20                                Unix
C:\TMP prog hello 10 20                            DOS
```

```
#include <stdio.h>
#include <stdlib.h>     /* for atoi and exit */
#include <string.h>

int main(int argc, char *argv[])         /* could also write char **argv */
{
    int n;
    int i;
    char **p;

    printf("The command line contains %d words.\n", argc);
    printf("There are %d command line arguments.\n", argc - 1);
    printf("The name of this program is %s.\n", argv[0]);

    if (argc > 1) {
        printf("The first command line argument is %s.\n", argv[1]);
        printf("The first command line argument contains %u characters.\n",
            strlen(argv[1]));

        if (argv[1][0] == '-') {         /* Handout 6, p. 8 */
            printf("The first command line argument starts with a dash.\n");
        }

        if (strcmp(argv[1], "bye") == 0) {
            return EXIT_FAILURE;
        }
    }

    if (argc > 2) {
        printf("The second command line argument is %s.\n", argv[2]);
        sscanf(argv[2], "%d", &n);   /* K&R p. 246; cf. sprintf in Handout 6, p. 2 */
        n = atoi(argv[2]);           /* another way to do same thing, K&R p. 251 */
        if (n > 100) {
            printf("The second command line argument is greater than 100.\n");
        }
    }

    /* Print the command line, one word per line. */
    for (i = 0; i < argc; ++i) {
        printf("%s\n", argv[i]);
    }

    /* Another way to write the same loop. */
    for (p = argv; p < argv + argc; ++p) {
        printf("%s\n", *p);
    }

    return EXIT_SUCCESS;
}
```

▼ **Homework 8.1: check the command line arguments**

Write a program that does two or more different things, depending on its command line arguments. Make sure an argument exists before doing anything with it. For example, make sure **`argc > 3`** before you mention **`argv[3]`**.

You get no credit if you ignore the **`else if`** instructions in Handout 1.

▲

Three kinds of scope for variables

(1) Variables that are function arguments (e.g., **`n`** in the **`line`** function) and variables declared inside the curly brackets that enclose a function (e.g., **`i`** in the first version of the **`line`** function) can be used only within that function. See K&R p. 31; King pp. 185−188.

(2) Variables declared **`static`** before the first function of a **`.c`** file can be used by all the functions in that file but no others. For example, to make an array that can be used only by two functions **`push`** and **`pop`**, put the two functions in a file by themselves and declare the array at the top of the file with the **`static`** keyword. These two functions are shown on K&R p. 77; King p. 188; K&R p. 83; King pp. 185−186 tells how to add the **`static`** keyword.

(3) Non-**`static`** variables declared before the first function of a **`.c`** file can be used in any function of the program, including functions in other **`.c`** files. There must also be an **`extern`** declaration for the variable at the start of every other **`.c`** file where the variable is used.

If the variable is to be used in many other **`.c`** files, write its **`extern`** declaration in a **`.h`** file and **`#include`** the **`.h`** file in all the other **`.c`** files. It is harmless to **`#include`** the **`.h`** file in the original **`.c`** file as well.

Four kinds of storage classes for variables

(1) Variables that are function arguments and variables declared inside the curly brackets that enclose a function (except for the ones declared to be **`static`**) are born when their function is called and die when their function returns. This may happen many times as the program runs. If the declaration for one of these

variables contains an initialization, the initialization happens anew each time the variable is reborn. Otherwise the variable is reborn with an unpredictable value.

```
void f(void)
{
    int i = 10;  /* initialized to 10 whenever f is called */
    int j;       /* initialized to an unpredictable value whenever f is called */
}
```

(2) Variables declared to be **static** inside the body of a function† and all variables declared before the first function of a **.c** file live throughout the lifetime of a program. In other words, a value you store in one of these variables will remain there until you replace it with a different value. These variables are all initialized to 0 unless you initialize them to something else. The initialization happens only once, at the start of the program. For example, here are some external variables.

```
#include <stdio.h>
int i = 5;   /* initialized to 5 once, at the start of the program. */
int j;       /* initialized to 0 once, at the start of the program. */

int main()
{
```

(3) To allocate and free a block of memory whenever you want, **#include <stdlib.h>** and call the functions **malloc**, **realloc**, and **free**.

(4) To allocate a block of memory in Unix which will outlive the program and which can be used by other programs, call the *shared memory* functions **shmget**, **shmat**, **shmdt**, **shmctl**, and **shmop**. Shared memory is outside the scope of this operating-system independent course.

Excerpts from stdlib.h

In some versions of C, the declaration for the **malloc** function is written in **malloc.h** instead of **stdlib.h**.

Since the return value of **malloc** is of data type **void ***, it can be stored into any pointer.

```
typedef unsigned int size_t;            may be a different data type on your machine

void *malloc(size_t n);
void *realloc(void *p, size_t n);
void free(void *p);
```

Dynamic memory allocation: K&R pp. 167−168, 252; King pp. 359−361

You must declare an array with a *constant expression:* a number or an expression consisting only of numbers (K&R p. 38). You can’t use a variable.

```
char a[12];             /* legal */
char a[12 + 1];         /* legal */
char a[i];              /* illegal */
```

What do you do if the dimension has to be a *variable expression:* one containing variables? Pass the dimension to **malloc** to dynamically allocate a block of memory. The return value of **malloc** is the address of the start of the block. Store the return value in a pointer variable so you can access the block as if it were a big array.

† See the paragraph on K&R p. 83 just before Exercise 4−11; King pp. 186−187.

I use the format **"%u"** (see Handout 2) to **printf** and **scanf** a variable of data type **size_t**, because on my machine **size_t** is really just another name for **unsigned int** as shown above. You may have to use a different format (e.g. **"%lu"**) on your machine. See the last bullet on K&R p. 260.

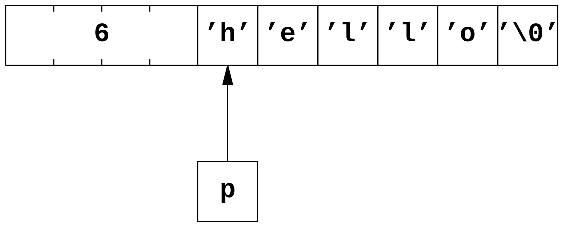
Ironclad rule 1. Whenever you call **malloc**, save its return value in a pointer variable. Never change the value of the pointer variable. Don’t even **++** the pointer variable. Keep the pointer variable unchanged, because later you will have to give it to the **free** function or the **realloc** function.

Ironclad rule 2. Whenever you call **realloc**, save its return value in a pointer variable. Never change the value of the pointer variable. Don’t even **++** the pointer variable. Keep the pointer variable unchanged, because later you will have to give it to the **free** function or another call of the **realloc** function.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6     size_t n;                        /* number of char’s in the block */
 7     char *p;                         /* hold address of first char in the block */
 8
 9     printf("How many char’s do you need in your array?\n");
10     scanf("%u", &n);
11
12     p = malloc(n);
13     if (p == NULL) {
14         printf("Can’t allocate %u bytes.\n", n);
15         return EXIT_FAILURE;
16     }
17
18     p[0] = ’h’;                      /* the first byte in the block */
19     p[1] = ’e’;
20     p[2] = ’l’;
21
22     /* etc. */
23
24     p[n-1] = ’\0’;                   /* the last byte in the block */
25
26     printf("%s\n", p);
27     printf("%u\n", ((size_t *)p)[-1]);
28
29     free(p);                         /* when you’re done with the block */
30     return EXIT_SUCCESS;
31 }
```

You can combine lines 12−13 to

```
32     if ((p = malloc(n)) == NULL) {
```



To **malloc** a block that will be used to hold **int**'s, make **p** a "pointer to **int**" and multiply the number of **int**'s by the size in bytes of each **int**:

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

int main()
{
    size_t n;                          /* number of int's in the block */
    int *p;                            /* hold address of the first int in the block */
    int i;                             /* should really be size_t */

    printf("How many int's do you need in your array?\n");
    scanf("%u", &n);
    p = malloc(n * sizeof(int));
    if (p == NULL) {
        printf("Can't allocate room for %u int's.\n", n);
        return EXIT_FAILURE;
    }

    p[0] = 31;                         /* the first int in the block */

    /* etc. */

    p[n-1] = 31;                       /* the last int in the block */

    for (i = 0; i < n; ++i) {
        printf("%d\n", p[i]);
    }

    free(p);
    return EXIT_SUCCESS;
}
```

▼ Homework 8.2: how much can you malloc? (not to be handed in)

What is the largest number of bytes you can request of **malloc** before it gives you **NULL**? Is this number (approximately) a power of 2? It's approximately 3,249,000,000 on i5.nyu.edu (up from 134,000,000 on our previous machine, acf5.nyu.edu, and 67,000,000 on the one before that, acf4.nyu.edu).

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

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

main()
```

```
 5 {
 6     size_t n;
 7     char *p;
 8 
 9     for (;;) {
10         printf("How many bytes should I allocate?\n");
11         scanf("%u", &n);
12 
13         p = malloc(n);
14 
15         if (p == NULL) {
16             printf("Can’t allocate %u bytes.\n", n);
17         } else {
18             printf("%u bytes were available.  Try for more.\n", n);
19             system("ps -o vsz,comm | awk ’NR == 1 || $2 == \"a.out\"’");
20             free(p);
21         }
22     }
23 }
```

▲

The definition for a convenient function

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

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 
 4 /* Allocate and return a pointer to n consecutive bytes of memory. */
 5 
 6 void *mymalloc(size_t n)
 7 {
 8     void *p = malloc(n);
 9 
10     if (p == NULL) {
11         printf("Can’t allocate %u bytes.\n", n);
12         exit(EXIT_FAILURE);
13     }
14 
15     return p;
16 }
```

You can split line 8 into

```
17     void *p;
18     p = malloc(n);
```

—but why would you want to? Or to obscure the code, you can change lines 8−10 to

```
19     void *p;
20 
21     if ((p = malloc(n)) == NULL) {
```

just like the familiar

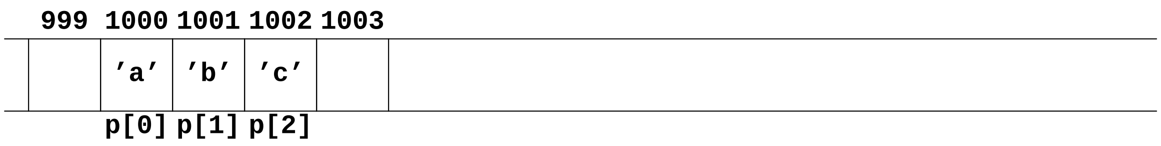
```
22     while ((c = getchar()) != EOF) {
```

Make the block larger: K&R p. 252; King 367−368

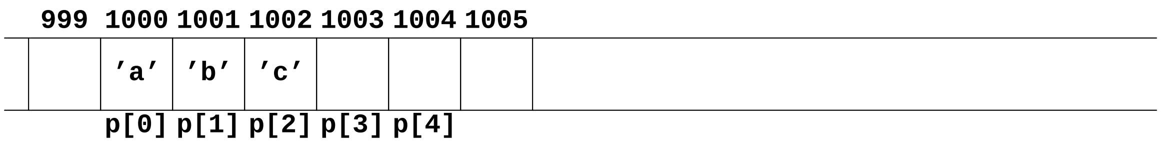
Our first `malloc` program allocated a block of `n` bytes of memory. If you need to enlarge the block later, call `realloc`. The first argument of `realloc` is the address of the start of the existing block. The second argument of `realloc` is the desired new size of the block in bytes, *not* the number of bytes that you want to append to the block.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6     char *p = malloc(3);
 7     if (p == NULL) {
 8         printf("Can’t allocate 3 bytes.\n");
 9         return EXIT_FAILURE;
10     }
11
12     p[0] = ’a’;              /* the first byte in the block */
13     p[1] = ’b’;
14     p[2] = ’c’;              /* the last byte in the block */
15
16     p = realloc(p, 5);
17     if (p == NULL) {
18         printf("Can’t allocate 5 bytes.\n");
19         return EXIT_FAILURE;
20     }
21
22     /* At this point, p[0], p[1], p[2] still contain ’a’, ’b’, ’c’. */
23     p[3] = ’d’;
24     p[4] = ’\0’;             /* the last byte in the block */
25     printf("%s\n", p);
26
27     free(p);
28     return EXIT_SUCCESS;
29 }
```

The situation at line 15:



`p` may still have its original value at line 21:



Or by line 21 `p` may have received a new value from the `realloc`:

1999	2000	2001	2002	2003	2004	2005
	'a'	'b'	'c'			
	p[0]	p[1]	p[2]	p[3]	p[4]	

Input a line of unpredictable length

```
/* Let the user type in a line of any length, and then print it out. */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    size_t i = 0;    /* number of characters the user has typed, plus 1 for
                        the terminating '\0'. */
    char *p;         /* hold the address of 1st byte of allocated block */
    char *q;
    int c;           /* each char read from input */

    /* Create a 1-byte block big enough to hold the terminating
    '\0' even if there are no input characters. */

    p = malloc(++i);
    if (p == NULL) {
        printf("Can't allocate %u bytes.\n", i);
        return EXIT_FAILURE;
    }

    printf("Please type a line and press RETURN.\n");

    while ((c = getchar()) != EOF && c != '\n') {
        p[i-1] = c;          /* Put c into the last byte of the block. */
        q = realloc(p, ++i);
        if (q == NULL) {
            printf("Can't allocate %u bytes.\n", i);
            return EXIT_FAILURE;
        }
        p = q;
    }
    p[i-1] = '\0';           /* Put '\0' into the last byte of the block. */

    printf("%s\n", p);
    free(p);
    return EXIT_SUCCESS;
}
```

A convenient function

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

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

char *mygetline(void);
```

```

int main()
{
    char *p;

    printf("Please type one line and press RETURN.\n");
    p = mygetline();

    printf("%s\n", p);
    free(p);
    return EXIT_SUCCESS;
}

/* Let the user input a line of any length.  Allocate a block of memory for the
line, excluding the RETURN that the user typed to terminate it.  Store the line
in the block, including a terminating ’\0’, and return the address of the first
char of the block. */

char *mygetline(void)
{
    size_t i = 0;
    char *p;
    char *q;
    int c;

    p = malloc(++i);
    if (p == NULL) {
        printf("Can’t allocate %u bytes.\n", i);
        exit(EXIT_FAILURE);
    }

    while ((c = getchar()) != EOF && c != ’\n’) {
        p[i-1] = c;
        q = realloc(p, ++i);
        if (q == NULL) {
            printf("Can’t allocate %u bytes.\n", i);
            exit(EXIT_FAILURE);
        }
        p = q;
    }

    p[i-1] = ’\0’;
    return p;
}
```

▼ Homework 8.3: consolidate the realloc ’s

```
const int initial = 10;  /* # of bytes to allocate initially */
const int increment = 3; /* # of additional bytes in each subsequent enlargement*/
```

Write these two **`const int`**’s immediately before the first line of the definition of the function **`mygetline`**. You get no credit if you write the numbers **`10`** and **`3`** elsewhere in the program. If you change the value of one or both of these two **`const int`**’s (e.g., if you make **`initial`** 20) and recompile, the program must still work correctly without any other change.

Instead of allocating one byte before the loop begins, allocate **`initial`** bytes. Allocate no additional memory during the first **`initial-1`** iterations. Then allocate **`increment`** additional bytes during one out of every **`increment`** subsequent iterations.

For example, the two **`const int`**'s shown above must cause the following allocations. Before the loop begins, you will give **`malloc`** an argument whose value is 10. Assuming that the user types a sufficiently long string, you will call **`realloc`** only on the tenth, thirteenth, sixteenth, nineteenth, etc., iterations. The first time you call **`realloc`**, you will give it a second argument whose value is 13. The second time you call **`realloc`**, you will give it a second argument whose value is 16. The third time you call **`realloc`**, you will give it a second argument whose value is 19; etc.

If the user types 11 or 14 or 17, etc., characters and then presses **`RETURN`**, you will therefore allocate one more byte more than you actually use to store the characters and the **`'\0'`**. If the user types 10 or 13 or 16, etc., characters and then presses **`RETURN`**, you will allocate two more bytes than you actually use to store the characters and the **`'\0'`**. If the user types 6 characters and then presses **`RETURN`**, you will allocate three more bytes than you actually use. Only if the user types 12 or 15 or 18, etc., characters and then presses **`RETURN`** will you use every byte that you allocate to store the characters and the **`'\0'`**.

The variable **`i`** performs a double duty in the above program: it counts how many characters the user has typed, and it is given as an argument to **`malloc`** and **`realloc`** to specify how many bytes to allocate. Let **`i`** continue in its first rôle, but create one new variable

```
    size_t n;            /* size in bytes of allocated block */
```

to be the argument of **`malloc`** and **`realloc`**. We will always have **`i`** ≤ **`n`**. The words **`malloc`** and **`realloc`** must each appear in only one place as in the above program; no credit otherwise. You get no credit if you store a dead value into a variable: see Handout 1, p. 17.

```
    int i = 0;          /* This 0 is a dead value: it's never used. */

    i = 1;
    printf("%d\n", i);
```

▲

Divide a program into several .c files: K&R pp. 80−81; King pp. 309−313

You don't have to write all the functions of a program in a single **`.c`** file. Here is how to define the **`main`** function in **`file1.c`** and the functions **`f`** and **`g`** in a separate file called **`file2.c`**.

You may optionally write the word **`extern`** at the start of any of the four function declarations below.

```
/* This file1.c. */
#include <stdio.h>
#include <stdlib.h>

void f(void);                    /* function declaration */
void g(void);                    /* function declaration */

int main()                           /* function definition */
{
    f();
    g();

    return EXIT_SUCCESS;
}

/* This is file2.c. */
#include <stdio.h>
```

```

void f(void);                     /* function declaration */
void g(void);                     /* function declaration */

void f(void)                      /* function definition */
{
    printf("This is function f.\n");
}

void g(void)                      /* function definition */
{
    printf("This is function g.\n");
}
```

```
1$ gcc file1.c file2.c
```

Write the function declarations in a .h file: K&R pp. 81−82; King pp. 307−308

Instead of writing the same lines at the start of each `.c` file, write them only once in a separate `.h` file. Put this `.h` file in the directory that contains the `.c` files of your program.

```
/* This file is prog.h. */
void f(void);
void g(void);
```

```
/* This file1.c. */
#include <stdio.h>
#include <stdlib.h>
#include "prog.h"

int main()
{
    f();
    g();

    return EXIT_SUCCESS;
}
```

```
/* This is file2.c. */
#include <stdio.h>
#include "prog.h"

void f(void)
{
    printf("This is function f.\n");
}

void g(void)
{
    printf("This is function g.\n");
}
```

```
1$ gcc file1.c file2.c
```

Don't mention the `.h` *file.*

▼ Homework 8.4: package the stack in a separate file

Put the stack in a separate **`.c`** file for either the program in Handout 7, pp. 17−18 or Homeworks 7.7 or 7.8. Create the following **`.h`** file which can be **`#include`**’d in any program that wants to **`push`** and **`pop`**.

```
/* This file is stack.h. */

void push(int n);
int pop(void);
```

Then create the following **`.c`** file which can be linked to any program that wants to **`push`** and **`pop`**.

```
/* This file is stack.c. */

#include "stack.h"
#define MAXVAL 100

static int val[MAXVAL];
static int sp = 0;

/* Now write the definitions for the functions push and pop. */
```

The C++ language gives you a more explicit notation for creating groups of variables which can be accessed only by specific functions. This group of variables, together with the functions that are allowed to access them, is called a *class.*

▲

Use a variable in two different .c files: K&R pp. 80−81; King pp. 309−310

To use a variable in several **`.c`** files, define it at the top of *one* **`.c`** file without the words **`extern`** and **`static`**. Write the initial value (default, **`0`**) in this definition:

```
int x = 10;                        /* variable definition */
```

At the top of every other **`.c`** file in which the variable is used, declare it to be **`extern`** without an initialization:

```
extern int x;                      /* variable declaration */
```

It is unnecessary to write a copy of the declaration above the definition, but do it anyway:

```
extern int x;                      /* variable declaration */
int x = 10;                        /* variable definition */
```

Now that you have written the same declaration at the top of every **`.c`** file in which the variable **`x`** is mentioned, you can remove the declaration from every **`.c`** file and write it in a **`.h`** file instead:

```
/* This file is prog.h. */
extern int x;
```

Compile a C or C++ program

i5 has the following compilers:

`cc`	*C*
`gcc`	*GNU C*
`g++`	*GNU C++*

We’ll use **`gcc`** as our example throughout. Give only the name of the **`.c`** file, not the names of the **`.h`** files, as a command line argument to **`gcc`**.

```
1$ gcc prog.c                      Create an executable file named a.out
2$ ls -l a.out                     a.out  has its x bits turned on.
3$ a.out                           Execute the C program.
4$ mv a.out prog                   Rename the a.out file prog

5$ gcc -o prog prog.c              Create an executable file named prog
6$ ls -l prog

7$ gcc -o ˜/bin/prog prog.c
8$ ls -l ˜/bin
```

`${1%.c}` is the shellscript's first command line argument, with the trailing **`.c`** chopped off.

```
#!/bin/ksh

/usr/local/bin/gcc -o ˜/bin/${1%.c} $*
```

<Angle brackets> in an #include directive

The **`/usr/include`** directory and its descendants contain the **`.h`** files that will be **`#include`**'d by many programs: **`stdio.h`**, **`stdlib.h`**, **`math.h`**, etc. Do not specify the full path name of these **`.h`** files when you **`#include`** them. Enclose them in **<**angle brackets**>**, which will make the compiler automatically add **`/usr/include`** to the start of their names before searching for them. Never write a full pathname within angle brackets.

`CC include` files are in the directory **`/usr/include/CC`**.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>

1$ cd /usr/include
2$ ls -l | more
-rw-r--r--   1 root     bin         9606 Nov 23  2004 math.h
-rw-r--r--   1 root     bin        11853 Jan 21  2005 stdio.h
-rw-r--r--   1 root     bin         9296 Jan 21  2005 stdlib.h

3$ ls *.h | wc -l
     250

4$ cd sys
5$ pwd
/usr/include/sys

6$ ls -l | more
-rw-r--r--   1 root     bin        17028 Jan 25  2006 types.h
```

"Double quotes" in an #include directive

If you do not want the compiler to add **`/usr/include`** to the start of the **`.h`** file's name, use **"**double quotes**"** instead of **<**angle brackets**>**. For example, to **`#include`** a **`.h`** file that will be used by only a few C programs,

```
#include "moon.h"                                        /* relative pathname */
#include "/home1/m/mm64/46/moon/moon.h"                  /* full pathname */
```

If you’re always going to be in the directory that contains the **`.h`** file when you give the **`gcc`** command, you can write the relative pathname. Otherwise, write the full pathname.

The -I option of gcc

Suppose the file **`stdlib.h`** was in an unusual place on your machine, e.g., the directory **`/usr/exclude`** instead of **`/usr/include`**. You could change all of your C programs from

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

to

```
#include "/usr/exclude/stdlib.h"
```

but then you would have to change them back when you port the program to a new machine.

A better solution would be to keep the **`#include <stdlib.h>`**, but use the **`-I`** option of **`gcc`** to tell the C compiler to try to add another directory name in addition to **`/usr/include`** to the start of an angle-bracketed name.

```
1$ gcc -I/usr/exclude -o prog prog.c
```
no space after the **`I`**

You can write several **`-I`** options, each one naming another directory.

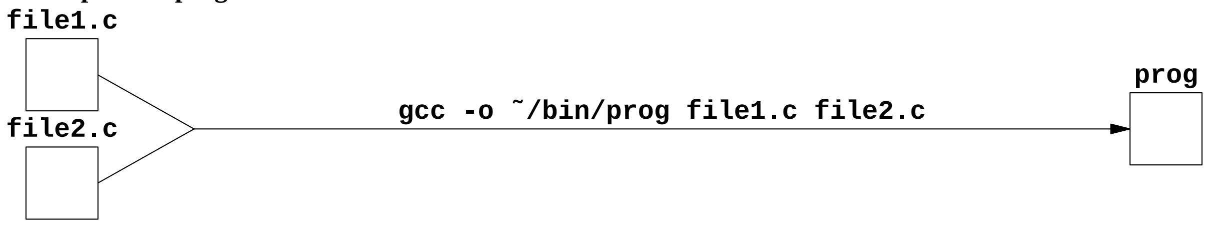
```
2$ gcc -I/usr/exclude -I.. -o prog prog.c
```
`..` *is current dir’s parent*

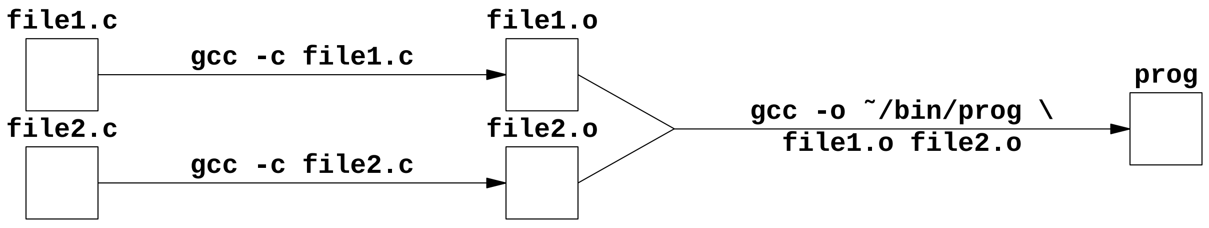
The **`-I`** option does not tell **`gcc`** where to find **`.c`** files; it tells **`gcc`** where to find only the angle-bracketed **`.h`** files.

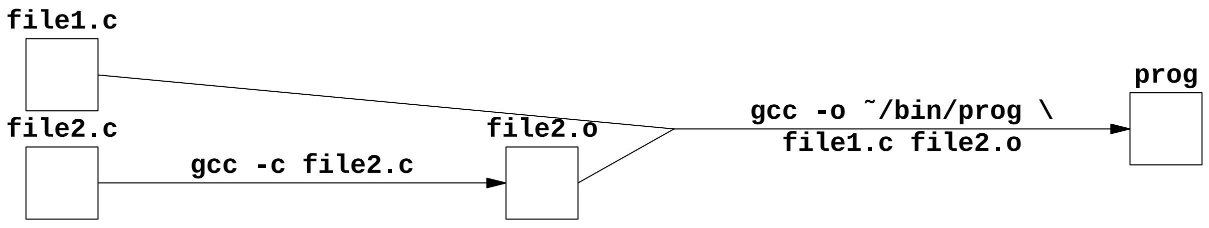
Intermediate steps in a C compilation

```
1$ gcc -E prog.c > prog.i          Create prog.i
2$ gcc -S prog.c                   Create prog.s
3$ gcc -c prog.c                   Create prog.o
```

Compile a C program divided into several .c files







Give the **`.c`** files as command line arguments to **`gcc`** in any order. Do not give the names of the **`.h`** files as command line arguments to **`gcc`**.

```
1$ gcc -o ~/bin/prog file1.c file2.c
```

Compile the **`.c`** files individually and then link them together:

```
2$ gcc -c file1.c                        Create file1.o
3$ gcc -c file2.c                        Create file2.o
4$ gcc -o ~/bin/prog file1.o file2.o
```

You can even give **`gcc`** a mixture of **`.c`** and **`.o`** files:

```
5$ gcc -c file2.c                        Create file2.o
6$ gcc -o ~/bin/prog file1.c file2.o
```

Libraries

If your program calls math library functions such as **`sqrt`**, **`sin`**, **`cos`**, etc., add the **`-lm`** option (minus lowercase LM) to the end of the **`gcc`** command.

When you specify a library with this **`-l`** option, the name of the file that contains the library is whatever follows the **`-l`**, with a **`lib`** added to the front and a **`.a`** added to the end. For example, the name of file that contains the library you specified with the **`-l`** option is **`libm.a`**.

By default, **`gcc`** assumes that every library is in the **`/usr/lib`** directory; you can specify a different directory with a **`-L`** option before the **`-l`** option.

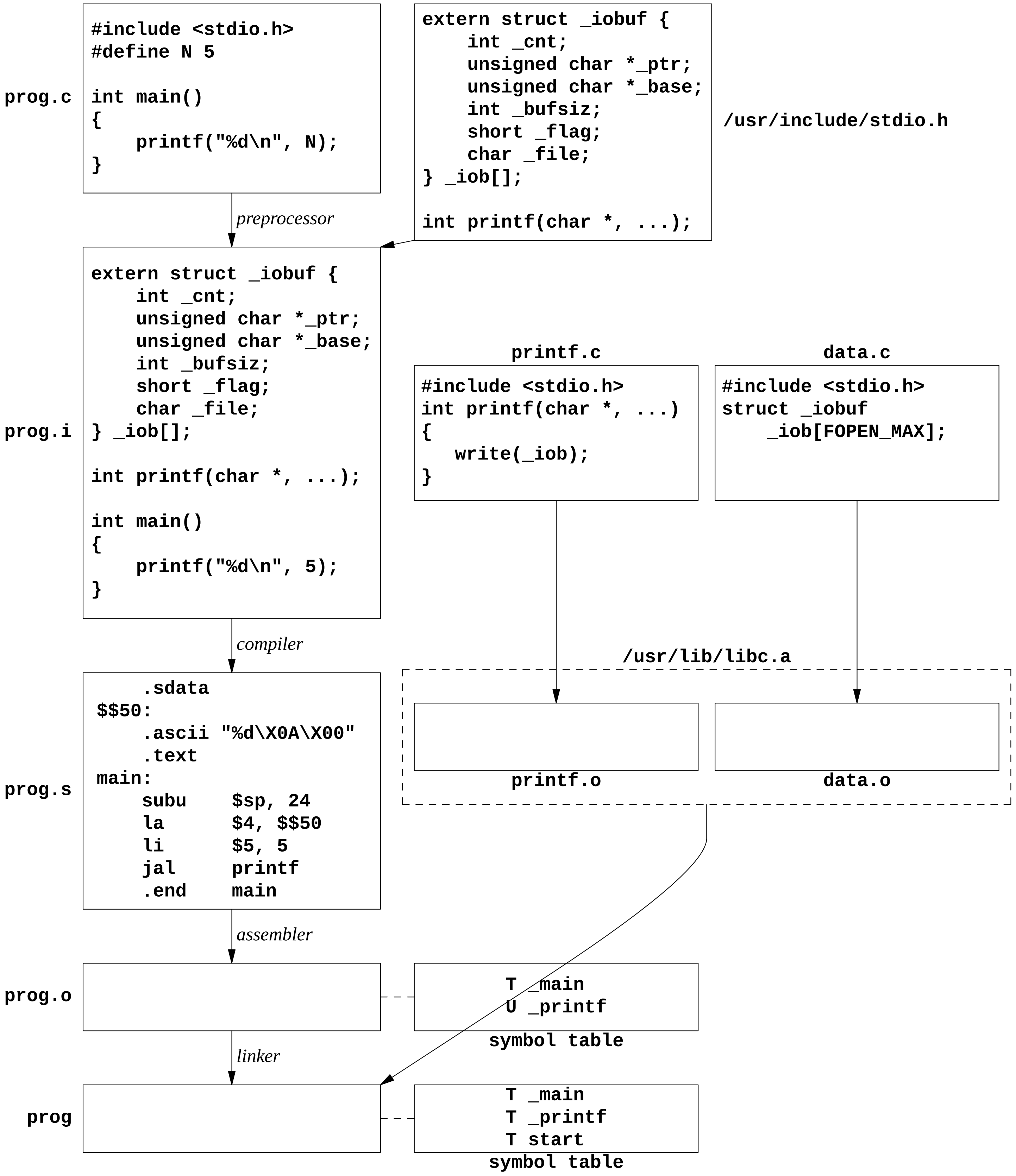
▼ Homework 8.5: create an executable moon

Copy the **`.h`** and **`.c`** files whose names start with **`moon`** from the directory **`$S32`** to your **`bin`** directory. Compile them into an executable file named **`moon`**. Put the **`-lm`** option (“math library”, minus lowercase LM) at the end of the **`gcc`** command line that links **`moon`** together.

```
1$ moon                          today
2$ moon 21 12 2006               day, month, year
```

▲

The file **/usr/include/stdio.h** contains the declarations for functions such as **printf** and **scanf**. See pp. 155−156 in K&R for declarations of functions that take a variable number of arguments.



prog.c:

```
#include <stdio.h>
#define N 5

int main()
{
    printf("%d\n", N);
}
```

/usr/include/stdio.h:

```
extern struct _iobuf {
    int _cnt;
    unsigned char *_ptr;
    unsigned char *_base;
    int _bufsiz;
    short _flag;
    char _file;
} _iob[];

int printf(char *, ...);
```

prog.i:

```
extern struct _iobuf {
    int _cnt;
    unsigned char *_ptr;
    unsigned char *_base;
    int _bufsiz;
    short _flag;
    char _file;
} _iob[];

int printf(char *, ...);

int main()
{
    printf("%d\n", 5);
}
```

printf.c:

```
#include <stdio.h>
int printf(char *, ...)
{
   write(_iob);
}
```

data.c:

```
#include <stdio.h>
struct _iobuf
    _iob[FOPEN_MAX];
```

prog.s:

```
    .sdata
$$50:
    .ascii "%d\X0A\X00"
    .text
main:
    subu    $sp, 24
    la      $4, $$50
    li      $5, 5
    jal     printf
    .end    main
```

prog.o symbol table:

```
T _main
U _printf
```

prog symbol table:

```
T _main
T _printf
T start
```

Differences between `if` and `#if`: K&R pp. 91−2; King pp. 288−289

The C preprocessor is a text editor, and **`#if`** is its "delete line(s)" instruction. It is a conditional compilation directive, like **`%IF`** in PL/I and **`AIF`** in IBM 360/370 assembler.

(1) The computer determines whether a **`#if`** is true or false *before* the program runs. Each **`#if`** is evaluated only once. On the other hand, the computer determines whether an **`if`** is true *as* the program runs. If an **`if`** is inside of a loop, it may be evaluated more than once.

(2) If the **`#if`** is true (i.e., the following expression is not zero), the **`#if`** line and the corresponding **`#endif`** line are ignored. If the **`#if`** is false (i.e., the following expression is zero), the **`#if`** line and the corresponding **`#endif`** line and all the lines between them are ignored.

(3) A **`#if`** can go anywhere in the program. An **`if`** can go only inside a function.

(4) A **`#if`** does not require parentheses around the logical expression, and has no **`{ }`**. It is terminated with a **`#endif`**. There are also **`#elif`** and **`#else`** directives.

(5) Because the computer determines whether a **`#if`** is true or false before the program runs, you can't use a variable in the logical expression. You can, however, use a macro created earlier with the word **`#define`**.

Conditionally compile debugging code with `#if`

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

#define DEBUG   1

int main()
{
    int debug;
    int i = 10;

    printf("Type 1 for debugging, 0 for no debugging: ");
    scanf("%d", &debug);

    if (debug == 1) {
        printf("i == %d\n", i);
    }

#if DEBUG == 1
    printf("i == %d\n", i);
#endif

    return EXIT_SUCCESS;
}
```

See **`<assert.h>`** on K&R p. 253 for a convenient way to insert conditionally compiled debugging statements.

Port the code to multiple platforms.

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

/* Give each of the following #define's a different number. */
#define UNIX      0
#define WINDOWS   1
#define MACINTOSH 2
```

```

#define MACHINE UNIX     /* which machine will this program run on */


#if MACHINE == UNIX
#define BUFFSIZE 2000
char name[] = "Unix";
#endif

#if MACHINE == WINDOWS
#define BUFFSIZE 1000
char name[] = "Windows";
#endif

#if MACHINE == MACINTOSH
#define BUFFSIZE 1000
char name[] = "Macintosh";
#endif

int main()
{
    char a[BUFFSIZE];

    printf("Welcome to the %s.\n", name);

#if MACHINE == WINDOWS
    printf("No one ever lost their job buying Microsoft.\n");
#endif

    return EXIT_SUCCESS;
}
```

Two ways to create a new name for a data type: K&R pp. 146−147; King pp. 129−131

The variables **`n1`**, **`n2`**, and **`n3`** hold catalog numbers of products at the Bongdex Compiler Corporation. Today there are 30,000 products but soon there will be 40,000. If the variables that hold catalog numbers had been **`int`**'s and if **`sizeof(int) == 2`**, we would then have to hunt down all of their declarations and change them from **`int`** to **`long`**:

```
int n1;
int n2;
int n3;
```

But far-sighted programmers declared them as **`catno_t`**'s instead of **`int`**'s, so that only one line will have to be changed.

```
#define catno_t int      /* Use either one, but not both, */
typedef int catno_t;     /* of these two lines. */

catno_t n1;
catno_t n2;
catno_t n3;
```

A **`typedef`** conventionally ends with **`_t`**. See **`size_t`** and **`ptrdiff_t`** in **`stddef.h`** (or **`stdlib.h`**) for other examples of **`typedef`**.

Why did they invent typedef?

In the above example, we can use either **`typedef`** or **`#define`** to create the new data type **`catno_t`**. But we can’t create the following new data type **`vision_t`** with a **`#define`** (unless we give the **`#define`** an argument), because the **`int`** and the **`[N]`** are not contiguous.

```
#define N 2                    /* left and right eyes */

int v1[N];                     /* v1 is an array of two int’s: v1[0] and v1[1]. */
int v2[N];
int v3[N];

#define N 2
typedef int vision_t[N];      /* another name for an array of N int’s */

vision_t v1;
vision_t v2;
vision_t v3;
```

```
typedef char airport_t[3];

const airport_t Kennedy   = {’J’, ’F’, ’K’};
const airport_t LaGuardia = {’L’, ’G’, ’A’};
const airport_t Newark    = {’E’, ’W’, ’R’};

    printf("%.3s\n", LaGuardia); /* .3 because LGA has no terminating ’\0’ */
```

Create a structure: pp. 127−132

```
 1 /* Create two variables called january and p. */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 int main()
 6 {
 7     struct {
 8         int length;                 /* number of days */
 9         double temperature;         /* average temperature in Fahrenheit */
10     } january, *p;                  /* january is a structure, p is a ptr to a struct *
11
12     january.length = 31;
13     january.temperature = 32.5;
14     printf("%d %f\n", january.length, january.temperature);
15
16     /* Another way to do the same thing.  Must give a value to
17     p before you can give a value to (*p).length or (*p).temperature */
18     p = &january;
19     (*p).length = 31;
20     (*p).temperature = 32.5;
21     printf("%d %f\n", (*p).length, (*p).temperature);
22
23     /* A better way to write lines 18-21. */
24     p = &january;
25     p->length = 31;
26     p->temperature = 32.5;
27     printf("%d %f\n", p->length, p->temperature);
```

```
    return EXIT_SUCCESS;
}
```

Two abbreviations

Just as the **!=** operator is an abbreviation for the **==** and **!** operators, and the **[]** operator is an abbreviation for the **+** and ***** operators (Handout 3, p. 10), the **->** operator is an abbreviation for the ***** and **.** operators. In both cases, the abbreviation makes the parentheses unnecessary.

before	*after*	
`!(a == b)`	`a != b`	
`*(p+i)`	`p[i]`	
`(*p).field`	`p->field`	
`(*(p+i)).field`	`(p+i)->field`	can combine the ***** and **.** to **->**
	`p[i].field`	but simpler to combine the **+** and ***** to **[]**

Never use the notations shown in the ‘‘before’’ column above.

Create a name for this type of structure

We create a name for this type of structure with the declaration on lines 7−10 below, and create a structure of this type with the declaration on line 16. I wrote the declarations for the variables **january** and **p** within the **main** function because I will use them only within that function. I wrote the declaration for the data type **struct month** at the top of the file so I could use **struct month** in other functions.

```
 1 /* Create a new data type named struct month,
 2 and several variables of this type. */
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6
 7 struct month {                /* Create a name for a new data type. */
 8     int length;               /* number of days */
 9     double temperature;       /* average temperature in Fahrenheit */
10 };
11
12 void f(void);
13
14 int main()
15 {
16     struct month january;     /* Create a variable of the new data type. */
17     struct month *p = &january;
18
19     p->length = 31;
20     p->temperature = 32.5;
21     printf("%d %f\n", p->length, p->temperature);
22
23     f();
24     return EXIT_SUCCESS;
25 }
26
27 void f(void)
28 {
29     struct month february;
30     /* etc. */
31 }
```

You can split line 17 to

```
17     struct month *p;
18     p = &january;
```

—but why would you want to?

Use typedef to create a name for this type of structure

Write a **`typedef`** declaration to create a name for your new data type. Then write separate declarations to create structures of this type. This will let you avoid having to type the word **`struct`** over and over again as in the above program.

```
 1 /* Create a new data type named month_t, and several variables of this type. */
 2
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 typedef struct {              /* Create a name for a new data type. */
 7     int length;               /* number of days */
 8     double temperature;       /* average temperature in Fahrenheit */
 9 } month_t;
10
11 void f(void);
12
13 int main()
14 {
15     month_t january;              /* Create a variable of the new data type. */
16     month_t *p = &january;
17     p->length = 31;
18     p->temperature = 32.5;
19     printf("%d %f\n", p->length, p->temperature);
20
21     f();
22     return EXIT_SUCCESS;
23 }
24
25 void f(void)
26 {
27     month_t february;
28     /* etc. */
29 }
```

You can split line 16 to

```
16     month_t *p;
17     p = &january;
```

—but why would you want to?

Create an array of structures: pp. 132−136

If you have only one month, don't bother to make it a structure. Simply declare two variables

```
    int length;              /* number of days */
    double temperature;      /* average temperature in Fahrenheit */
```

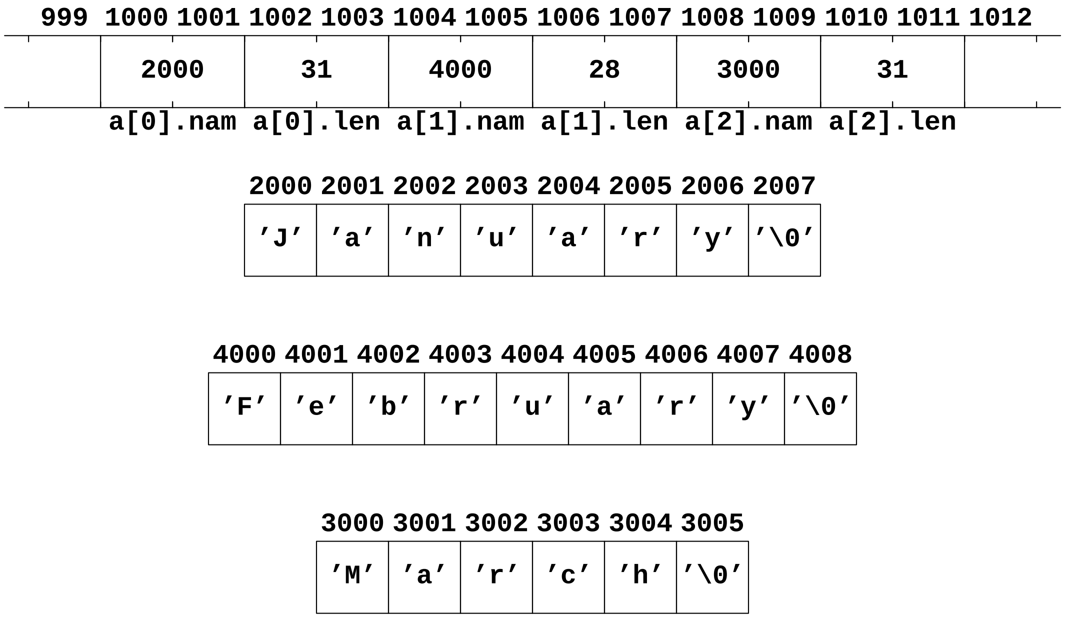
Create structures only when you have many months. The **{**curly braces**}** on lines 15−26 are optional; those on lines 14 and 27 are required.

```
 1 /* Print the number of days in the year and the temperature of the hottest
 2 month. */
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6
 7 typedef struct {
 8     int length;              /* number of days */
 9     double temperature;      /* average temperature in Fahrenheit */
10 } month_t;
11
12 int main()
13 {
14     month_t a[] = {
15         {31,    32.5},       /* January */
16         {28,    37.4},       /* February */
17         {31,    42.1},       /* March */
18         {30,    53.0},       /* April */
19         {31,    62.9},       /* May */
20         {30,    74.1},       /* June */
21         {31,    79.3},       /* July */
22         {31,    74.7},       /* August */
23         {30,    69.9},       /* September */
24         {31,    57.0},       /* October */
25         {30,    48.9},       /* November */
26         {31,    34.0}        /* December */
27     };
28 #define N (sizeof a / sizeof a[0])       /* number of months */
29
30     int m;
31     int sum = 0;
32     double hottest = a[0].temperature;
33     month_t *p;
34
35     for (m = 0; m < N; ++m) {
36         sum += a[m].length;
37         if (a[m].temperature > hottest) {
38             hottest = a[m].temperature;
39         }
40     }
41
42     /* Another way to write the same loop. */
43     for (p = a; p < a + N; ++p) {
44         sum += p->length;
45         if (p->temperature > hottest) {
46             hottest = p->temperature;
47         }
48     }
49
```

```
    printf("There are %d days in the year.\n", sum);
    printf("The average temerature of the hottest month is %f.\n", hottest);
    return EXIT_SUCCESS;
}
```

An array of structures with a string field

Assume that **`sizeof(char *) == 2`** and **`sizeof(int) == 2`**. Each element of the array therefore occupies 4 bytes:



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

typedef struct {        /* structure to hold information about one element */
    char *name;         /* name of the month */
    int length;         /* number of days */
} month_t;

int main()
{
    month_t a[] = {
        {"January",      31},
        {"February",     28},
        {"March",        31},
        {"April",        30},
        {"May",          31},
        {"June",         30},
        {"July",         31},
        {"August",       31},
        {"September",    30},
        {"October",      31},
        {"November",     30},
        {"December",     31}
    };
25 #define N   (sizeof a / sizeof a[0])
26
27     int i;
28     month_t *p;
29     int sum = 0;
30
31     for (i = 0; i < N; ++i) {
32         printf("%-9s %d\n", a[i].name, a[i].length);
33         sum += a[i].length;
34     }
35
36     /* Another way to write the same loop. */
37     for (p = a; p < a + N; ++p) {
38         printf("%-9s %d\n", p->name, p->length);
39         sum += p->length;
40     }
41
42     printf("There are %d days in a year.\n", sum);
43     return EXIT_SUCCESS;
44 }
```

What not to store in a structure

It would be redundant to store consecutive numbers in a column of an array of structures. In line 34 change **`a[i].number`** to **`i`**, and in line 44 change **`p->number`** to **`p-a`**. Then delete line 7 and the column of numbers that initializes the **`number`** field.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 typedef struct {
 6     char *name;         /* name of element */
 7     int number;         /* atomic number */
 8     double weight;      /* atomic weight */
 9 } element_t;
10
11 int main()
12 {
13     element_t a[] = {
14         {NULL,        0,    0.0},    /* Dummy, so subscript == atomic number */
15         {"hydrogen",  1,    1.0080}, /* Column two is a waste of space. */
16         {"helium",    2,    4.0026},
17         {"lithium",   3,    6.9390},
18         {"beryllium", 4,    9.0122},
19     };
20 #define N   (sizeof a / sizeof a[0])
21
22 #define BUFFSIZE 256     /* number of bytes in input buffer */
23     char input[BUFFSIZE];
24
25     int i;
26     element_t *p;
27
28     printf("Please type an element and press RETURN.\n");
```

```
    scanf("%s", input);

    for (i = 1; i < N; ++i) {
        if (strcmp(a[i].name, input) == 0) {
            printf("%s has atomic number %d and atomic weight %.4f\n",
                a[i].name, a[i].number, a[i].weight);
            return EXIT_SUCCESS;
        }
    }
    return EXIT_FAILURE;

    /* Another way to write the same loop. */
    for (p = a + 1; p < a + N; ++p) {
        if (strcmp(p->name, input) == 0) {
            printf("%s has atomic number %d and atomic weight %.4f\n",
                p->name, p->number, p->weight);
            return EXIT_SUCCESS;
        }
    }
    return EXIT_FAILURE;
}
```

Examples of structure `typedef`'s

```
typedef struct {
    int math;                /* in the range 200 to 800 inclusive */
    int verbal;              /* in the range 200 to 800 inclusive */
} sat_t;                     /* Scholastic Aptitude Test */

typedef struct {
    int systolic;            /* contract: bigger number */
    int diastolic;           /* relax: smaller number */
} pressure_t;                /* blood pressure */

typedef struct {
    double bust;             /* in inches */
    double waist;            /* in inches */
    double hips;             /* in inches */
} measurements_t;
```

▼ Homework 8.6: rewrite using an initialized array of structures

Do only one of the following four programs. In each case, create a new data type with a **`typedef struct`**. Then create and initialize an array of this new data type.

(1) Rewrite Homework 6.3 to use an initialized array of structures. Write only one **`printf`** in a **`for`** loop. Loop through the array with a pointer to each element instead of an integer index. The elements of the array will be

```
planet_t a[] = {
    {"Mercury",   .27},
    {"Venus",     .85},
    {"Earth",    1.00},       /* etc. */
```

(2) Rewrite the example in Handout 5, p. 3 to use one initialized array of structures instead of two parallel initialized arrays. The variable **`p`** will now be a pointer to a structure; you will need no other

pointers. Remove the variables **i** and **q**, and remove the loops in lines 18-27.

(3) Rewrite the answer to Homework 6.4.3 to use one initialized array of structures instead of two parallel initialized arrays. Don't use a pointer to a structure; use the month number as an integer index. To simplify the program, make a dummy structure as the first array element:

```
month_t a[] = {
    {NULL,       0},      /* dummy element to give January subscript 1 */
    {"January",  31},
    {"February", 28},
    {"March",    31},     /* etc. */

    if (strcmp(a[month].name, input) == 0) {
```

(4) Write a C program with an initialized array of any kind of structure. Do something to the array: e.g., loop through it searching for something, or print out its entire contents, or change some of the values of its fields. Examples of tabular material suitable for storage in an array of structures include the Köchel catalog of Mozart's works, the periodic table of the elements, 501 Spanish Verbs, the NYU course catalog, the discography of the Grateful Dead, the novels of Sinclair Lewis, the notes of the scale (giving their letter names and vibrations per second), the ex-Soviet republics (name, area, population; see **/home1/m/mm64/46/data/ussr**), the planets (name, distance from sun, diameter, length of year [which is a function of the distance from sun]), the stars on the Main Sequence (spectral type and temperature—*oh, be a fine girl, kiss me*), ordinary people (name, address, and social security number), or the words to a repetitious song or poem.

Write a program for historians of the French Revolution that will convert dates from the revolutionary calendar (e.g., 15 Thermidor of the year II) to our calendar. Write a program that will convert Jewish, Islamic, or Chinese dates to our calendar, at least for this year.

For example, here is part of the initialization for an array of structures in a program that will make change. (Use **%** for remainder.) *Do not write the names and values of the coins anywhere except in the array of structures.* This will let us change to a foreign currency by changing only the array of structures, not the rest of the program.

```
        {1,     "penny",    "pennies"},
        {5,     "nickel",   "nickels"},
        {10,    "dime",     "dimes"},    /* etc. */
```

The user will type in a number of cents, and the program will print the shortest list of coins equal to that amount. For example, if the user types 82 cents, the program will print

```
    3 quarters
    1 nickel
    2 pennies
```

Convert Roman numerals to Arabic numerals:

```
        {'I',   1},     /* first field is a char */
        {'V',   5},
        {'X',   10},    /* etc. */
```

Print the words to "Old MacDonnald had a Farm":

```
        {"pig",      "oink"},
        {"chicken",  "cluck"},
        {"cow",      "moo"},
```

Ask for the user's sign and tell their horoscope, or ask for their birthday and compute their sign:

```
        {"Aries",    "Ram",   "Make time to read some poetry today."},
        {"Taurus",   "Bull",  "Put your fist through a CRT this afternoon."},
        {"Leo",      "Lion",  "Try using an odd number of parentheses."},
```

Or make Bond trivia quiz:

```
        {1962,  60000000,    "Sean Connery",      "Dr. No"},
        {1963,  79000000,    "Sean Connery",      "From Russia With Love"},
        {1964, 125000000,    "Sean Connery",      "Goldfinger"},
        {1965, 141000000,    "Sean Connery",      "Thunderball"},
        {1967, 111000000,    "Sean Connery",      "You Only Live Twice"},
        {1969,  65000000,    "George Lazenby",    "On Her Majesty’s Secret Service"},
        {1971, 116000000,    "Sean Connery",      "Diamonds Are Forever"},
        {1973, 126000000,    "Roger Moore",       "Live and Let Die"},
```

Convert foreign currency to US currency and vice versa. There are actually two different rates, depending on the direction, so create an array each of whose elements is a structure containing two `double`’s in addition to the name of the country and the name of the currency.

Write the simplest possible program that will do the job. You will be graded on how easy your program is to understand, not on how big it is. Did you accidentally declare any variables that you didn’t use? Can you eliminate statements or temporary variables by reordering your statements? Will looping in a different direction make the loop easier to start or stop? Is there any repetitious code which can be eliminated or banished to an array? That’s what arrays are for.

▲

Dynamically allocate memory for all the strings in an array of strings

The `char *a[N]` in the following program is not big enough to hold `N` strings. It holds only the address of the first character of each string—the strings themselves are stored elsewhere in additional memory. To make room to store each string in memory, it therefore does not suffice to merely declare

```
    char *a[N];
```

For each string you must also call `malloc` to allocate the additional memory. The call to `malloc` (and `realloc`) is in `mygetline`.

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

```
/* Let the user input N strings.  Allocate exactly enough memory to
hold a copy of each string, including the terminating ’\0’.  The array
a[] holds the address of each of these N blocks. */

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

#define N   10      /* how many strings */

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

    printf("Please type %d lines and press RETURN after each.\n", N);

    for (i = 0; i < N; ++i) {
        a[i] = mygetline();
        printf("%s\n", a[i]);
    }

    /* Another way to do the same thing. */
    for (p = a; p < a + N; ++p) {
        *p = mygetline();
        printf("%s\n", *p);
        free(*p);
    }

    return EXIT_SUCCESS;
}
```

Dynamically allocate memory for all the strings in an array of structures

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

```
/* Let the user input N structures.  Each structure contains a pointer to a
string--the string is not stored in the structure itself.  Allocate exactly
enough memory to hold a copy of each string, including the terminating '\0'. */

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

#define N   10      /* how many structures */

typedef struct {
    char *name;
    int n;
} animal_t;

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

    printf("Please type a number and a name, %d times,\n", N);
    printf("and press RETURN after each number and each name.\n");

    for (i = 0; i < N; ++i) {
        scanf("%d", &a[i].n);
        a[i].name = mygetline();
        printf("%d %s\n", a[i].n, a[i].name);
        free(a[i].name);
    }

    /* Another way to do the same thing. */
    for (p = a; p < a + N; ++p) {
        scanf("%d", &p->n);
        p->name = mygetline();
        printf("%d %s\n", p->n, p->name);
        free(p->name);
    }

    return EXIT_SUCCESS;
}
```

□