The Art of C programming

Compilers and Interpreters

> **Interpreter**: > "when the program is executed, to take each line in turn, translate it into machine code and then run the machine code. When the translation of a line takes place, the resulting machine code overwrites that from the previous line, so that if a line of source code appear in a loop which is executed 200 times, it also must be translated 200 times"
>
> **Compilers**: > "The obvious alternative is to translate the entire source code into a single machine code program, and then run the machine code. This will clearly execute faster, and there are no translations during execution. A translator that adopts this approach is called a compiler."

The Skeleton of a C program

> "All C programs consist of a series of functions"
>
> "The effective length of a function name is often six letters"
>
> "In C, the newline character has no significance at all"

Loop and Control Constructs

```c
/* conditional expression */
if (x == 0)
    printf("x is zero");
/* is exactly equivalent to */
if (!x)
    printf("x is zero");

/* logical connectives */
if (a > 10 && a < 20)
    printf("a is within range");
/* this could be negated: */
if (!(a > 10 && a < 20))
    printf("a is not within range");
```

Arithmetic and Logic

> "... and then the different branches of arithmetic – Ambition, Distraction, Uglification, and Derision". Alice's Adventures in Wonderland

ASCII coding system in binary

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

- Bit 7: Not used in standard ASCII
- Bit 6: 0=Digit, 1=Letter

- Bit 5: 0=Upper case, 1=digit/lower case
- Bit 4: 0= a-o, 1=digit/p-z

- Bit 3:

- Bit 2:

- Bit 1:

- Bit 0:

AND (&)

> "The AND operation compares two bit patterns and produces a third result pattern. If corresponding pairs of bits are both 1 the result bit for that position is set to 1. Under all other circumstances it is set to 0"

```
/* toupper, convert lower case to their upper case equivalents */
toupper(char c) {
    char mask;
    mask = 223;
    return (c & mask);
}
```

Binary value of 223: `11011111` bit 5 is forced to zero, and the other bits are left unchanged.

> "a bit that is set, i.e., 1, is considered"true"; a cleared bit, or 0, is"false". Thus, in the result of `z = x & y`, each bit is set if and only if the corresponding bit is set in both `x` and `y`."

OR

> "If either of the corresponding pairs of bits is a `1` the result bit for that column is `1`. Only if both bits are zero is the result zero"

```
p   01101010
q   00100011
p|q 01101011
```

```
/* tolower: lower case letters from either upper of lower case arguments/*
tolower(char c){
    char mask;
    mask = 32;  /* 00100000 */
    return (c | mask);
}

/* you can see that bit 5 is forced to 1 */
```

XOR This is short for eXclusive OR. It's like OR except that it excludes the condition that both bits are '1'.

```
p   01101010
q   00100011
p^q 01001001

1^1 = 0
```

Now we can write the function swapcase

```
/* swapcase: converts lower to upper case and vice versa */
swapcase(char c) {
    char mask;
    mask = 32;
    return (c ^ mask);
}
```

NOT

> "There's one remaining logical operator, NOT. This inverts every bit in a field."

```
p   01110001
~p  10001110
```

Shifts

> "If you shift a bit pattern left and fill in to the right with zeros it has the effect of multiplying by 2"

```
p   00010011 = 19
<<  00100110 = 38

/* this can be done directly with */
y = x << 1;
/* will shift x left 1 bit and put the result in y */

/* similary: */
p = r << 4;
```

```
/* would shift r 4 bits left, resulting in a multiplication by 16 */
```

Here a "mulitply by ten" technique:

```
times10(int n) {
    int m, p;
    m = n << 1;
    p = m << 2;
    return (m + p);
}
```

Right shifts are also possible: `a = b >> 2;`, would shift b 2 bits right and put the result in a.

> "Normally, a right shift is equivalent to a divide by 2, as you'd expect, but there are exceptions to this rule which we'll come across later."

Strings, Arrays, and pointers

Strings and Pointers

Number	Character
38012	t
38013	h
38014	i
38015	s
38016	
38017	m
38018	e
38019	s
38020	s
38021	a
38022	g
38023	e
38024	0

0 = [ASCII null delimiter]

When C compiles a statement like `s = "this message";` it sets up a set of consecutive bytes somewhere in memory [...]. You'll notice that there's a delimiter provided by the system, which is a zero byte. That's not ASCII zero (whose value is 48) but ASCII null (i.e., all bits set to '0').

The variable `s` is set to `38012`. In other words, it is not the string but rather a pointer to the string. For the moment it is enough to note that it is pointer to strings, rather than the strings themselves, that are passed around in a C program.

Even in the `"printf("some output");` the argument that is really passed to printf is the pointer to the string "some output".

Arrays

```
char s[30];

/* transfer the string "some stuff" to the array s */
strcpy(s, "some stuff");
```

> An array name is a pointer to the beginning of the array. However, there is one difference: an array name is a constant and you can't do arithmetic with it whereas a pointer can be manipulated in any way you like.

String Functions

Let's illustrate pointer manipulation by writing string functions. So first we want a function that is passed a pointer to the string, and will return the number of bytes the string contains:

```
/* Writing String functions that mimic BASIC's LEN, LEFT$, RIGHT$, and MID$ */

/* len: return string length */
int len(char s[]) {
    int k;
    k = 0;
    while (s[k++])
        ;
    return (k-1);
}

/* left: copy the lefmost 'n' char from string to sub array */
void left(const char string[], char sub[], int n) {
    int i;
    for (i = 0; i < n && string[i] != '\0'; i++) {
        sub[i] = string[i];
    }
    sub[i] = '\0';
}

/* right: copy rightmost n char */
void right(const *string, char *sub, int n) {
    int length = strlen(string);
    int start = length - n; /* starting index for copying */

    /* ensure we don't start before beginning of the string */
```

```
        if (start < 0)
            start = 0;

        int i, j;
        for (i = start, j = 0; i < length; i++, j++) {
            sub[j] = string[i];
        }
        sub[j] = '\0';
    }

    /* substring: takes a string and two numeric arguments: one for
     * starting position and one for the number of char to transfer */
    void substring(const char *string, char *sub, int start, int length) {
        int string_length = strlen(string);

        /* ensure starting index is within bounds */
        if (start < 0 || start >= string_length) {
            sub[0] = '\0';
            return;
        }

        /* ensure length doesn't exceed */
        if (start + length > string_length) {
            length = string_length - start;
        }

        int i;
        for (i = 0; i < length && string[start + i] != '\0'; i++)
            sub[i] = string[start + i];
        sub[i] = "\0';
    }

}
```

More about Pointers

Let's revise the len function using pointer.

```
    int len(char *s) {
        char *begin;
        begin = s;
        while (*s++)
            ;
        return (s - begin - 1);
    }
```

This is clearly a very close relative of the original len function. We're just missing

a few square brackets, and we have to store the initial pointer value rather than setting a counter to zero.

```c
/* let's write strcpy */
/* strcpy: copy move string about from source to target */
char *strcpy(char *dest, const char *src) {
    char *dest_start = dest;
    while (*src)
        *dest++ = *src++;
    *dest = '\0';
    return dest_start;
}

/* strcat: concatenate two strings together */
char *strcat(char *string, const char *extra) {
    char *dest_strat = string;
    string += strlen(string);
    while (*extra)
        *string++ = *extra++;
    *string = '\0';
    return dest_start;
}
```

Projects:

```c
/* Write a function day0 which takes a pointer *date pointing to a string
* giving a date, in the format "March 18 1937", and returns a pointer *day
* to a string that gives the corresponding day of the week
*/

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

const char *months[] = {
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
};
const char *days_of_week[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
    "Saturday"
};

int month_to_number(const char *month) {
    for (int i = 0; i < 12; i++) {
        if (strcmp(months[i], month) == 0) {
            return i + 1;
```

```
        }
    }
    return -1; /* month not found */
}

const char *day0(const char *date) {
    char month_str[20];
    int day, year;
    sscanf(date, "%s %d %d", month_str, &day, &year);

    int month = month_to_number(month_str);
    if (month == -1) return NULL;

    /* adjust month and year for January and February */
    if (month == 1 || month == 2) {
        month += 12;
        year --;
    }

    /* Zeller's Congruence Algorithm
     * Used to determine the day of the week for any given date
     *
     * K = the year within the century (1937 % 100 = 37)
     * J = century (2001 / 100 = 20)
     */

    int K = year % 100;
    int J = year / 100;

    /* Zeller's Congruence formula */
    int f = day + (13*(month+1))/5 + K + K/4 + J/4 + 5*J;
    int day_of_week = f%7;

    day_of_week = (day_of_week + 6) % 7;

    return days_of_week[day_of_week];
}

int main() {
    const char *date = "March 18 1937";
    const char *day = day0(date);
    if (day != NULL) {
        printf("The day of the week for %s is %s.\n", date, day);
    else {
        printf("Invalid date format.\n");
    }
```

```
    return 0;
}

/* Write a perpetual calendar which, given the month and year, prints
 * out which days of that month fall on which dates, in the usual
 * calendar format:
 *
 * August                                          1986
 * SUN   MON     TUE     WED     THU     FRI     SAT
 *                                       1       2
 * 3     4       5       6       7       8       9
 * 10    11      12      13      14      15      16
 * [...]
 * 24    25      26      27      28      29      30
 * 31
 *
 */

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

const char *months[] = {
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
};

const char *days_of_week[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
    "Saturday"
};

int month_to_number(const char *month) {
    for (int i = 0; i < 12; i++) {
        if (strcmp(months[i], month) == 0)
            return i + 1;
    }
    return -1; /* month not found */
}

const char *day0(const char *date) {
    char month_str[20];
    int day, year;
    sscanf(date, "%s %d %d", month_str, &day, &year);

    int month = month_to_number(month_str);
```

```
    if (month == -1) return NULL;

    /* adjust month and year for January and February */
    if (month == 1 || month == 2) {
        month += 12;
        year--;
    }

    /* Zeller's Congruence Algorithm */
    int K = year % 100;
    int J = year / 100;

    /* Zeller's Congruence formula */
    int f = day + (13*(month+1))/5 + K + K/4 + J/4 + 5*J;
    int day_of_week = f % 7;

    day_of_week = (day_of_week + 6) % 7;
    return days_of_week[day_of_week];
 }

 int days_in_month(int month, int year) {
    if (month == 2) {
        /* check for leap year */
        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0) {
            return 29;
        else {
            return 28;
         }
        }
    }
    if (month == 4 || month == 6 || month == 9 || month == 11) {
        return 30;
    }
    return 31;
}

void print_calendar(int month, int year) {
    char first_date[20];
    /* snprintf: write to the array of char (string) with max n size of characters */
    snprintf(first_date, sizeof(first_date), "%s 1 %d", months[month-1], year);
    const char *first_day = day0(first_date);
    int start_day = -1;
    for (int i = 0; i < 7; i++) {
        if (strcmp(days_of_week[i], first_day) == 0) {
            start_day = i;
            break;
```

```
        }
    }
    if (start_day == -1) {
        printf("invalid date.\n");
        return;
    }
    printf("%s %d\n", months[month-1], year);
    printf("SUN MON TUE WED THU FRI SAT\n");

    int days = day_in_month(month, year);
    int day = 1;

    for (int i = 0; i < start_day; i++)
        printf("    ");

    for (int i = start_day; i < 7; i++)
        printf("%-5d", day++)
    printf("\n");

    while (day <= days) {
        for (int i = 0; i < 7 && day <= days; i++)
            printf("%-5d", day++)
        /* -: left-aligned, 5 char wide, d: deciaml */

        printf("\n");
    }
}

int main() {
    int month, year;
    printf("Enter month (1-12): ");
    scanf("%d", &month);
    printf("enter year: ");
    scanf("%d", &year);

    if (month < 1 || month > 12) {
        printf("invalid month. enter a value between 1 and 12);
        return 1;
    }

    print_calendar(month, year);
    return 0;
}
```

Floats and Other types

> "She can't do Subtraction", said the White Queen. "Can you do Division? devide a loaf by a knife - what's the answer to that?"

The unsigned form allows you to tell the compiler that you want to think about the number as always being positive, and you get a doubled maximum value as an advantage.

Declaring Structures

```
/* definition of a cat_entry */
struct cat_entry {
    int item_no;
    char description[30];
    float price;
    int stock_level;
};

/* catalog is an array of 200 elements, each of which has the
 * the cat_entry structure
 */
struct cat_entry catalog[200];

/* cat_entry has become just another type
 * Here a pointer to objects of that type
 */
struct cat_entry *p;
/* you can now use p to point anywhere in the catalog */
```

Defining Your Own Types

Defining structures comes close to defining completely new types. Well, you can do that, too, with a typedef statement.

```
/* declares int and month to be equivalent */
typedef int month;
/* so now you can write */
month date1, date2;
/* date1 and date2 are declared to be of type month and so are ints */

/* better yet, cat_entry above could be given a synonym: */
typedef struct cat_entry catalog;
/* and then */
catalog unipart[5000], great_universal_stores[2000];
```

None of this provides you with any new programming tools. What it does achieve is a readability improvement.

Constants and Initializers

The exponent form for large or small numbers is allowed

```
float very_big_number;
very_big_number = 1.3e9;
/* will allocate 1,300,000,000 to the variable */
```

Character Constants

We've spent some time looking at string constants, and you might imagine that a C character can be seen simply as a string of length 1. However, this isn't so, because a string of length 1 occupies 2 bytes, the second containing the null delimiter. So we need a way of allocating a value to a single byte.

```
char c;
c = 'A';
```

In other words, when you use a single quotes marks you are setting the value of a single bytes; double quote marks identify a string, as we've seen before.

Handling Control Characters

Control characters are those that have some effect (on a printer, say) but don't actually print a symbol. And if you can't print it, how can you put it between quote marks?

However, there are some control characters that are so commonly needed that C makes special provision for them. Each is preceded by a backslash, called, in the jargon, an *escape*.

Here's the complete set:

Escape Sequence	Description
`\n`	newline
`\t`	tab
`newline`	newline
`tab50`	tab
`\b`	backspace
`\r`	carriage return
`\f`	form feed
`\\`	backslash
`\'`	single quote
`\ddd`	octal number
`\0`	ASCII null

Octal and Hecadecimal Constants

There are times when a constant, although numeric, doesn't *really* represent a number and isn't a character either. Mask patterns of logical operations are good examples. That because is the *bit patterns* rather than the numbers they represent which are significant. So in cases like these, it would be nice to have format that translates to binary more easily than decimal does. C allows us to declare constants in either base 8(octal) or base 16(hexadecimal). In the former case, each digit can be directly encoded from a group of 3 bits, and in the latter case, groups of 4 bits.

Octal	Binary	Hexadecimal	Binary
0	000	0	0000
1	001	1	0001
2	010	2	0010
3	011	3	0011
4	100	4	0100
5	101	5	0101
6	110	6	0110
7	111	7	0111
		8	1000
		9	1001
		A	1010
		B	1011
		C	1100
		D	1101
		E	1110
		F	1111

So, for example, the pattern `10000001` can be seen as `10  000 001` giving `2  0  1` in octal or `1000  0001` giving `8  1` in hexadecimal

That work because three binary digits can represent exaclty one octal digit, reflecting the fact that three binary places cover the full range of a single octal digit.

C needs a way to distinguish between decimal, octal, and hexadecimal constants. If there is a leading zero, the constant is taken as octal. If there is a leading zero followed by an 'x' the constant is hexadecimal.

```
/* 10000001 */
mask = 129;
mask = 0201;
mask = 0x81;

/* are all equivalent statements */
```

Initializers

It's possible to initialize the value of a variable when you declare it:

```
int t = 7;
/* is equivalent to */
int t;
t = 7;

/* an array can be initialized in a similar way */
int discount[3] = {0,7,12};
/* is equivalent to */
int discount[3];
discount[0] = 0;
discount[1] = 7;
discount[2] = 12;
```

Initializing Pointers

```
int a[50], *p;
p = a;
/* will make p point to the beginning of the array a
 * because a is itself a pointer to the beginning of the array */

int fred, *p;
p = &fred;
/* p is now a pointer to fred. */

/* &a[0] = array name, here a */
int a[50], *p;
p = a;
/* is exactly equivalent to */
int a[50], *p;
p = &a[0];

/* if you want to search an array from top to bottom */
int a[50], *p, *q;
p = a;
q = &a[49];
while (*p++ != *q--)
        ;
/* checking whether there is a pair of symmetric elements
 * (one from the start and one from the end)
 */
```

Chapter 7: Input

> "The last time she saw them they were trying to put the dormouse in the teapot" – Alice's Adventures in Wonderland

Primitive Input At the simple level, C provides a function getchar which returns the next character input from the keyboard.

The most obvious extension to getchar is to allow a set of characters to be input to a buffer for subsequent processing. We'll write a function getbuf for this.

The function `getbuf` reads characters one by one from the input using `getchar()` and stores each character into the memory location pointed to by the buffer provided.

```
#define BUFSIZE 40
#define DELIM 13 /* ASCII code for carriage return */

/* here the first attempt of the getbuf function */
void getbuf(char *buffer) {
    char *bptr;
    for (bptr = buffer; bptr < buffer + BUFSIZE; bptr++) {
        *bptr = getchar();
        if (*bptr == DELIM)
            return;
    }
}

/* here the modern C style */
void getbuf(char *buffer) {
    char *bptr = buffer;

    for (; bptr < buffer + BUFSIZE; bptr++) {
        /* using int for getchar() to handle EOF properly */
        int ch = getchar();
        if (ch == EOF)
            break;

        *bptr = (char)ch;
        if (*bptr == DELIM)
            return;
    }
}

/* another attempt with a while loop */
void getbuf(char *buffer) {
    int num_chars;
```

```
    num_chars = 0;

    while ((*buffer++ = getchar()) != DELIM && num_chars < BUFSIZE)
        num_chars++;

    *buffer = '\0';
}
```

`EOF` is typically defined as `-1`. It is a special constant defined in `<stdio.h>`. This value is distinct from all valid `char` values, which ensures that it can be used to detect the end of input reliably.

Order of precedence

In the list of operator precedence, the operators that appear first have the highest precedence and are evaluated before those at the end.

```
(), [ ]
*(pointer), &(adress), -(negative), !, ~, ++, --
*(multiply), /, %
+, -(minus)
>>, <<
<, >, <=, >=
==, !=
&(bitwise and), ^, |
&&, ||
=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=
```

The highest the priority operators are those at the top of the list, and those on the same lines have equal priority.

```
/* Modify getbuf function so that it stores characters with ASCII codes
 * incremented by one compared to the input char.
 */

void getbuf(char *buffer) {
    int num_chars = 0;
    int ch;

    while (num_chars < BUFSIZE) {
        ch = getchar();
        if (ch == DELIM || ch == EOF) {
            break;
        }
        *buffer++ = ch + 1;
        num_chars++;
    }
```

```
    *buffer = '\0';
}
```

Strings to Numbers

> "My opinion is that you should never make things difficult until you've made them simple."

```
#include <ctype.h>      /* for isdigit function */
#include <stdbool.h>    /* for using bool, true, false */

int convert(const char *p) {
    int result = 0;
    int sign = 1;

    /* check for negative sign */
    if (*p == '-') {
        sign = -1;
        p++; /* move pointer to the next char */
    }

    /* convert string to integer */
    while (true) {
        if (!isdigit(*p)) {
            /* current char is not a digit, return the result */
            return result * sign;
        }
        result = result * 10 + (*p - '0');
        p++;
    }
}

/* NOTE:
 * isdigit() return true if *p is a digit, the ! operator negates
 * the result. Making 0 true, that the case when is not a digit.
 * The if block is then executed.
 */
```

This algorithm has two requirements for it to work correctly. First, as I've already pointed out, there must be a sign preceding the digits, and second, there must be at least one digit, because the first character in the string i added into result regardless of what it is.

This means that atoi must ensure that theses conditions exit before calling convert.

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

```c
#include <ctype.h>
#include <string.h>

#define BUFSIZE 1024

int convert(const char *num);
int setup(const char *buffer, char *num);

int custom_atoi(const char *buffer) {
    /* there's a chance that we need one more byte for the sign */
    char num[BUFSIZE + 1];
    if (!setup(buffer, num)) {
        printf("Not an integer\n");
        exit(EXIT_FAILURE);
    }
    return convert(num);
}

int setup(const char *buffer, char *num) {
    const char *start = buffer;

    /* skip non-digit char at beginning */
    while (!isdigit(*buffer)) {
        if (!*buffer) /* if we reach EO String without finding a digit */
            return 0;
        buffer++;
    }

    /* check if the char before is a negative sign */
    if (buffer > start && *(buffer-1) == '-') {
        *num = '-'; /* add the nagative sign */
    }
    /* cpy the digits */
    while (isdigit(*buffer)) {
        *num++ = *buffer++;
    }

    *num = '\0';
    return 1;
}

int convert(const char *num) { ... }
```

"In C, characters are internally represented as integers based on their ASCII values"

```c
char c = 'A';       /* 'A' has an ASCII value of 65 */
```

```c
int i = 65;

/* %zu specifier to print values of type size_t */
printf("size of char: %zu bytes\n", sizeof(c)); /* 1 bytes */
printf("size of int: %zu bytes\n", sizeof(i));  /* 4 bytes */

printf("char c as integer: %d\n", c); /* output -> 65 */
```

```c
int isdigit(char c) {
    if (c >= '0' && c <= '9')
        return 1;
    else
        return 0;
}

/* That clear in this case that c is representing a digit */
```

A Feeble Excuse

> "Getchar isn't the only way to grab things from the outside world. There's a function called scanf which does the job in a way that mirrors the operation of printf"

```c
/* Substitution cypher program.
 * On first running, this should request as input a permutation of
 * the alphabet that will be used instead of the original one.
 * String of 26 uppercase letters needed as an input.
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define ALPHABET_LENGTH 26

/* create_mapping: create the substitution cipher mapping */
void create_mapping(char mapping[], const char permutation[]) {
    for (int i = 0; i < ALPHABET_LENGTH; i++) {
        mapping[i] = permutation[i];
    }
}

/* encode_message: encode a msg using the mapping */
void encode_message(char mapping[], const char message[], char encoded[]) {
    int len = strlen(message);
    for (int i = 0; i < len; i++) {
        if (isalpha(message[i])) {
```

```
            if (isupper(message[i])) {
                /* map uppercase letters */
                encoded[i] = mapping[message[i] - 'A'];
            }
            else {
                /* map lowercase */
                encoded[i] = tolower(mapping[message[i] - 'a']);
            }
        }
        else {
            /* non-alpha char remain unchanged */
            encoded[i] = message[i];
        }
    }
    encoded[len] = '\0';
}

int main() {
    char permutation[ALPHABET_LENGTH + 1];  /* for null-terminator */
    char mapping[ALPHABET_LENGTH];
    char message[256];
    char encoded[256];

    /* input permutation */
    printf("Enter a permutation of alphabet: ");
    scanf("%s", permutation);

    /* create the mapping */
    create_mapping(mapping, permutation);

    /* loop to encode multiple msgs */
    while (1) {
        printf("Enter a message to encode (or type EXIT to quit): ");
        /* read line of input, and skip leading whitespace */
        scanf(" %[^\n]", message);

        if (strcmp(message, "EXIT") == 0)
            break;

        /* encode msg */
        encode_message(mapping, message, encoded);

        /* output the encoded msg */
        printf("Encoded message: %s\n", encoded);
    }
```

```
    return 0;
}
```

> "When `scanf` encounters a space, it knows that is should skip over any whitespace characters before processing the next input according to the next specifier"

By default `scanf` skips leading whitespace when using format specifiers like `%s` or `%d` , but does not skip them when using custom format specifiers like `"%[^\n]"` (or `%c`)

Output

> "The first thing she heard was a general chorus of 'There goes Bill!' then the Rabbit's voice alone—'Catch him, you by the hedge!' then silence, and then another confusion of voices—'Hold up his head—Brandy now—Don't choke him—How was it, old fellow? What happened to you? Tell us all about it!'" *Alice's Adventures in Wonderland*

We'll now turn our attention to more ways of getting the machine to talk to us...

Printf format specifiers:

- o: convert to octal
- x: convert to hexadecimal
- u: convert to unsigned decimal
- s: print as a string
- f: convert to decimal (from floating point)

```
printf("%s%6.3f", p, x);
```

If `p` is a pointer to the string "First value is:" and `x` holds the value 5.813, you'll get the display `First value is: 5.813`

Notice that I've specified the length of the floating point value (6 char, 3 of them decimal places), but left C to make up its own mind about the length of the string. This clearly presents no problem, because C knows where the string ends from the terminating null. Howevern C will convert the floating point number to a string before printing it, so the same should be true for the `f` conversion (and, indeed, all the others).

Example with `3.43` with `%6.3f` - `.3f`: print the float with 3 digits after the decimal point, giving `3.430` - `6`: specifies the minimum field width. The total width of the printed number, including the digits and decimal point, is 5 char, since the minimum field width is 6, an extra space will be added to the left.

In other words the number is pushed as far right as possible (i.e., right justified). It is possible to force left justification by preceding the length specifier with a

minus sign like this `printf("%s%-6.3f", p, x);`

Printing to Memory There's a function sprintf that acts like printf except that the data it assembles isn't ouput at all, but passed to a string.

```
void printmoney(int cents) {
    char out[8] = " ";  /* init empty char array of size 8 */
    *out = '$';         /* first char to $ */
    /* 5 char wide, right-jus, start from out[i] */
    sprintf(out + 1, "%5d", cents);
    /* next 3 lines modify the string to insert a decimal point */
    *(out + 6) = *(out + 5);    /* cpy char at pos 5 to pos 6 */
    *(out + 5) = *(out + 4);    /* cpy char at pos 4 to pos 5 */
    *(out + 4) = '.';           /* insert decimal point at pos 4 */
    printf(out);
}
```

The junior two digits are shifted right to allow space for the '.' to be inserted.

```
out[0] = $
out[1] = (space)
out[2] = 3
out[3] = 8
out[4] = 4
out[5] = 5
out[6] = 5 (modified later)
out[7] = \0 (implicitly added by sprintf)
```

So you can see how a string can be manipulated in a very flexible way, whereas a straight printf would not have allowed such tinkering.

Primitive Output

We could write a function called type that makes the computer behave as a typewriter until a delimiter(reffered to as DELIM) is encountered.

```
void type() {
    char c;
    while((c = getchar()) != DELIM)
        putchar(c);
}
```

If you could change the standard input to a disc file, for example, we could use this routine as it stands to copy file to the screen. For that we need to alter the standard I/O devices. We shall return to this problem in Chapter 12.

More about Control Constructs

> "At the end of three yards I shall repeat them –for fear of your forgetting them. At the end of four, I shall say good-bye. And at the end of five, I shall go!"

Write more compact code using the conditional operator

```
int abs(value) {
    return (value < 0 ? -value : value);
}

int is_even(int number) {
    return (number % 2 == 0) ? 1 : 0;
}
```

Leaping out of Loops

```
for (i = 0; i < 100; i++) {
    if (!*p)
        break;
    *q++ = *p++;
}
```

The keyword `break` has the effect of breaking out of the smallest enclosing loop. It doesn't matter what kind of loop it is.

Continuing

It's also possible to jump to the end of a loop without leaving it. In other words you can conditionally bypass a chunk of the code in the loop.

```
/* For example:
 * Let bypass any zeros in the source array from the above code
 */
for (i = 0; i < 100; i++) {
    if (!*p)
        break;
    if (*p++ == '0')
        continue;
    q++ = *(p-1);
}
```

> "Experience gives you a feel for what construct is mort natural in a given circumstance"

Multiway Switches

C provides a mechanism for dealing directly with multiway branches, called the switch statement.

The `default` keyword doesn't have to be the final case. In fact, there is no significance at all in the order in which the case appear.

Do not pass Go

About the only situation where a 'goto' would make sense to use would be a series of nested loops at the center of which is a confition, which, if true, requires the whole block of loops to be exited. Using break in that case wouldn't make sense, because it only breaks out of the smallest enclosing loop.

Chapter 10: Recursion

> "Among the first things for which children are berated when they are learning their mother tongue is any attempt at a circular definition."A chair is a chair with. . . ","You can't explain something by using the word you're explaining!" they're told firmly.
>
> "Actually, with certain restrictions, circular definitions are perfectly legitimate, and they provide a powerful tool for the programmer. An equivalent of a circular definition is a function which calls itself."

```c
/* conventional loop */
int factorial(int n) {
    int result;
    result = 1;
    while (n--)
        result *= n;
    return result;
}

/* recursive way */
int factorial(int n) {
    if (n == 1)
        return 1;
    return n*factorial(n-1);
}
```

`char *p[200];` p is a contiguous block of memory that can hold 200 pointers.

Chapter 11: Structures

> "When I use a word." Humpty-Dumpty said. in a rather scornful tone. "it means just what I choose it to mean- neither more nor less . . .. When I make a word do a lot of work . .. I always pay it extra."
> – Through the Looking Glass

Playing with Structures

```
struct cat_entry {
    int item_no;
    char description[30];
    float price;
    int stock_level;
};

struct cat_entry catalog[5000];

/* identify a strucure member */
structure_name.member
catalog[147].stock_level -= n;

/* C philosophy is, to refer to their element with pointers */
pointer_to_strucutre_name->member;

/* declare pointer to struct cat_entry */
struct cat_entry *pcat;
/* then if pcat is pointing to catalog[147] */
pcat->stock_level -= n;
/* of course, pcat could have been set up with */
pcat = &catalog[147];
/* but it's more likely to have been determined by some search routine */
```

The Storeman's Mate

Let's pull some ideas together for dealing with comings and goings of catalog items.

- Add entry to catalog
- Delete entry from catalog
- Alter stock level
- Check price of item

We can use a switch on a function called menu which display the menu and returns the user's choice.

```
main() {
    int hell_frozen_over = 0;
    char menu();

    while (!hell_frozen_over)
        switch(menu()) {
            case '1': add();
                break;
            case '2": delete();
```

```
                break;
            case '3': alter_stock();
                break;
            case '4': check_price();
                break;
            default: exit(0);
        }
}

menu() {
    printf("options are\n");
    printf("1) Add Entry to Catalog\n");
    printf("2) Delete Entry from Catalog\n");
    printf("3) Alter Stock Level\n);
    printf("4) Check Price of Item\n");
    printf("Hit any other key to exit:")
    return (getchar());
}


/* delete: indicated being deleted by setting item_no field to zero
 * final record catalog[499] set to -1 to act as a delimiter
 */
 delete() {
    struct cat_entry *p;
    int target, new_item_no;
    p = catalog;

    printf("Enter item no for deletion:");
    scanf("%d", &target);
    while (p->item_no != target) {
        if (p->item_no < 0) {
            printf("Not found\");
            return;
        }
        p++;
    }
    printf("Confirm deletion of %s(y/n)", p->description);
    c = getchar();
    if (c  == 'y' | c == 'Y')
        p->item_no = 0;
 }

/* add: look for first zero item number, and shovel new data */
add() {
    struct cat_entry *p;
```

```
    p = catalog;

    while (p->item_no) {
        if (p->item_no) {
            printf("No room\n");
            return;
        }
        p++;
    }

    printf("Enter item number:");
    scanf("%d", &(p->item_no));

    /* check if item number already exists */
    struct cat_entry *check = catalog;

    while (check->item_no != -1) {
        if (check->item_no == new_item_no) {
            printf("item number taken!");
            return;
        }
        check++;
    }

    /* add new item */
    p->item_no = new_item_no;
    printf("Enter description:");
    scanf("%s", &(p->description));
    printf("Enter price:");
    scanf("%f", &(p->price));
    p->stock_level = 0;
}

/* alter_stock: alter stock level */
void alter_stock() {
    struct cat_entry *p;
    p = catalog;
    int target, new_stock;

    printf("Enter item no to alter stock: ");
    scanf("%d", &target);

    while (p->item_no != -1) {
        if(p->item_no == target) {
            printf("current stock lvl %d\n", p->stock_level);
            printf("enter new stock lvl: ");
```

```
            scanf("%d", &new_stock);
            p->stock_level = new_stock;
            printf("stock lvl updated.\n");
            return;
        }
        p++;
    }
    printf("item not found\n");
}

/* check_price: retrieve the price of any given entry */
void check_price() {
    struct cat_entry *p;
    p = catalog;
    int target;

    printf("enter item number to check his price: ");
    scanf("%d", &target);

    while (p->item_no != -1) {
        if (p->item_no == target) {
            printf("the price of %s is: %.2f\n", p->description, p->price);
            return;
        }
        p++;
    }
    printf("item not found\n");
}


/* print_catalog_items: print all item number */
void print_catalog_times() {
    struct cat_entry *p = catalog;
    while (p->item_no != -1) {
        if (p->item_no != 0)
            printf("%d", p->item_no);
        p++;
    }
}

/* find_key: returns a pointer to the appropriate record */
struct cat_entry* find_key(int target) {
    struct cat_entry *p = catalog;
    while (p->item_no != target) {
        if (p->item_no < 0)
            return 0;
```

```
        p++;
    }
    return p;
}
```

Recursive Structures

> "Here a recursive structure that consists of a pair of pointers, one of which points to a word, and the orther points to another structure of the same kind"

```
struct list {
    char *head;
    struct list *tail;
}
```

Projects

```
/* Write an additional menu functions that will order the entries by ascending item_n
 * sort: order entries in ascending item_no, bubble way
 */
void sort() {
    int count = 0;
    struct cat_entry *p = catalog;

    /* count active entries */
    while (p->item_no != -1) {
        if (p->item_no != 0)
            count++;
        p++
    }

    if (count <= 1)
        return;

    struct cat_entry temp;
    for (int i = 0; i < count-1; i++) {
        for (int j = 0; j < count-i-1; j++) {
            if (catalog[j].item_no > catalog[j+1].item_no) {
                temp = catalog[j];
                catalog[j] = catalog[j+1];
                catalog[j+1] = temp;
            }
        }
    }
    printf("catalog sorted by item number.\n");
```

```
}

/* print_catalog: display entire current catalog */
void print_catalog() {
    struct cat_entry *p = catalog;
    while (p->item_no != -1)
        printf("No: %d\n", p->item_no);
        p++;
    }
    printf("\n");
}

/* find_key: adding an argument that it will also accept a description
 * to search on
 */
struct cat_entry* find_key(int item_no, const char* description) {
    struct cat_entry *p = catalog;
    while (p->item_no != -1) {
        if ((item_no != -1 && p->item_no == item_no) ||
        (description != NULL && strcmp(p->description, description) == 0))
            return p;
        p++;
    }
    retrun NULL;
}

/* reoder: display all catalog entries whose stock level is below 20 */
void reorder() {
    struct cat_entry *p = catalog;
    while (p->item_no != -1) {
        if (p->stock_level < 20)
            printf("item no: %d have a stock below 20", p->item_no);
        p++;
    }
}
```

Chapter 12: File-Handling

I/O Redirection

Most modern operating systems do not concern themselves directly with peripheral devices such as keyboards and printers, but rather with notional entities called channels which are then associated with specific device drivers. Thus a program communicates with a channel and the channel is linked to a device. The advantage of this mechanism is that the program need not be changed if the source of its input data changes, or its output is to be transferred to a disk

file, say, rather than the printer. Only the channel/device assignment needs to be altered.

All C I/O is handled in this channel oriented way. Thus the getchar function does not (strictly) transfer a character from the keyboard to main memory; it transfers it from a channel, which by default is assigned to the keyboard, to main memory. This channel is referred to as stdin(for standard input). There is a corresponding channel stdout whose default assignment is the screen.

A typical mechanism for informing the operating system of this redirection of I/O might be `cprogram < a:data.text > b:newdata.text`

Error Messages

There is one difficulty with this arrangement: any runtime error messages would be redirected along with the output data and the results could be somewhat mystifying. To get over this problem there is a third standard channel called stderr to which error messages are sent.

Buffered File I/O

- `fopen` : open a channel to a file
- `getc` : get a character from a channel
- `putc` : send a character to a channel
- `fscanf` : like scanf, but access a channel rather than the keyboard
- `fprintf` : like printf, but access a channel rather than the screen
- `fclose` : flush a channel buffer and close a file

```c
/* creating a file */
int main() {
    FILE *cid;  /* channel id */
    int n;
    cid = fopen("square.dat", "w");
    for (n = 1; n < 101; n++)
        fprintf(cid, "%d %d\n", n, n*n);
    fclose(cid);
}

/* get the data back again */
    FILE *cid;
    int n, v, vsq;
    cid = fopen("squares.dat", "r");
    for (n = 1; n < 101; n++) {
        fscanf(cid, "%d %d", &v, &vsq);
        printf("%d %d", v, vsq);
    }
    fclose(cid);    /* free the channel identifier */
```

```c
/* filecopy utility */
FILE *in_cid, *out_cid;
char source[20], dest[20], c;
printf("File from:");
scanf("%s", source);
printf("File to:");
scanf("%s", dest);
in_cid = fopen(source, "r");
out_cid = fopen(dest, "w");
while ((c = getc(in_cid)) != EOF)
        putc(c, out_cid);
fclose(in_cid);
fclose(out_cid);

/* if you want a specific 'display file on screen'
 * you don't need to open an output file at all.
 * Just replace putc(c, out_cid); with putc(c, stdout);
 * or come to that: putchar(c);
 */
```

`fopen` tells us whether it was successful in opening file. It does this by returning a channel identifier value of zero if the file could not be opened for any reason.

Typically, `putc` and `fclose` also will report failure. EOF may be returnned by `putc`, and a non-zero value by `fclose`.

```c
/* So a more robust filecopy would be with those test included: */
if ((in_cid = fopen(source, "r")) == 0) {
    printf("No file %s", source);
    exit(0);
}

if ((out_cid = fopen(dest, "w")) == 0) {
    printf("Cannot open %s", dest);
    exit(0);
}


while ((c = getc(in_cid)) != EOF)
        if (putc(c, out_cid) == EOF) {
            printf("disk error");
            exit(0);
        }
}
```

Random Access Files

C provides a library function that effectively allows you to think about a file as

a character array on disk, and it provides a mechanism for setting the "array subscript".

The general form of this function is: `lseek(cid, skip_bytes, start);`

`0` if the skip is to be computed from the beginning of the file `1` if the skip is to be computed from the current position `2` if the skip is to be computed from the end of the file

For instance `lseek(cid, 200, 0);` will set the system up so that the next getc will read the 200th byte in the file.

A subsequent call `lseek(cid, 50, 1);` would arrange for byte 250 to be read nextd; or, of course, written next, with `putc`.

Leaving the file in "append" state with `lseek(cid, 0, 2);` which moves the subscript to the end of the file (start-2) and skipts 0 bytes.

```
/* using lseek to set up the position in the file */
int main() {
    FILE *in_cid;
    int n, skip, v, vsq;
    in_cid = fopen("square.dat", "r");
    printf("No. to be squared");
    scanf("%d", &n);
    skip = 10*(n-1);
    lseek(in_cid, skip, 0);
    fscanf(in_cid, "%d %d", &v, &vsq);
    printf("%d squared is %d", v, vsq);
}
/* fscanf is used to read the whole record */
```

Binary Search

Binary search algorithm can be stated like this:

> "Examine the middle entry in the file. If this is the target record, the job is done. If its key field is greater than the target field, the target record must lie in the bottom half of the file. Otherwise it must lie in the top half of the file. In either case half the file has been eliminated from the search, and the process is repeated, at each stage eliminating half the remaining entries."

```
/* display square root of a given number using binary search */
main() {
    FILE *in_cid;
    int top_rec, bottom_rec, mid_rec, key, square, skip, sqrt;
    top_rec = 100; bottom_rec = 1; key = 0;
    mid_rec = 50;
    in_cid = fopen("square.dat", "r");
```

```
    printf("Enter no. to be square rooted"); scanf("%d", &square);
    while (square != key) {
        skip = 10*(mid_rec - 1);
        lseek(in_cid, skip, 0);
        fscanf(in_cid, "%d %d", &sqrt, &key);
        if (key > square)
            top_rec= mid_rec - 1;
        else
            bottom_rec = mid_rec + 1;
        mid_rec = (top_rec + bottom_rec) / 2;
    }
    printf("square root is %d\n", sqrt);
}

/* write the function search_file which accpets the arguments
 * file name, number of record in the file, the record length,
 * the position of the first byte of the key, the length in bytes
 * of the key, and the target key.
 *
 * n = search_file("data", 2000, 25, 7, 3, test_int);
 * will look for the int "test_int" in bytes 7, 8, 9 of a 2000 record
 * file called "data", each of whose records is 25 bytes long.
 */

/* search_file: binary search to find a specific integer key within
 * each record */
int search_file(char *filename, int filesize, int recsize,
                int startbyte, int numbytes, int target) {
    FILE *in_cid;
    int top_rec, bottom_rec, mid_rec;
    int key = 0;    /* extracted key from current record    */
    int skip = 0;   /* byte offset to seek                  */
    char sub[numbytes+1];
    char record[recsize+1];

    top_rec = filesize;
    bottom_rec = 0;

    if ((in_cid = fopen(filename, "r")) == 0) {
        printf("no file %s", filename);
        return -1;
    }

    while (bottom_rec <= top_rec) {
        mid_rec = (top_rec + bottom_rec) / 2;
        skip = recsize*(mid_rec-1);
```

```
            lseek(in_cid, skip, 0);
            fscanf(in_cid, "%s", record);

            strncpy(sub, record + startbyte, numbytes);
            sub[numbytes] = '\0';
            key = atoi(sub);

            if (key == target) {
                fclose(in_cid);
                return mid_rec + 1;      /* record number (1-based index) */
            }
            else if (key > target) {
                top_rec = mid_rec - 1;
            }
            else {
                bottom_rec = mid_rec + 1;
            }
        }
        return -1;
    }
```

Projects:

```
    /* The binary search algorithm only work if the record are sorted into
    key order. Write a function that will sort the file into ascending
    order of any numeric keyfield */

    typedef struct {
        char *data;
        int key;
    } Record;

    int extract_key(const char *record, int startbyte, int numbytes) {
        char sub[numbytes + 1];
        strncpy(sub, record + startbyte, numbytes);
        sub[numbytes] = '\0';
        return atoi(sub);
    }

    int compare_records(const void *a, const void *b) {
        Record *recA = (Record *)a;
        Record *recB = (Record *)b;
        return (recA->key - recB->key);
    }

    void swap(Record v[], int i, int j) {
```

```
    Record temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

/* qsort: using lomuto partition */
void qsort(Record v[], int left, int right) {
    int i, last;
    if (left >= right)
        return;
    swap(v, left, (left+right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (compare_records(&v[i], &v[left]) < 0)
            swap(v, ++last, i);

    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}

void sort_file(char *filename, int filesize, int recsize,
               int startbyte, int numbytes) {
    FILE *in_cid;
    int num_records = filesize / recsize;
    Record *records = malloc(num_records * sizeof(Record));

    if ((in_cid = fopen(filename, "r")) == NULL) {
        printf("no file %s\n", filename);
        free(records);
        return;
    }

    /* read records into the array */
    for (int i = 0; i < num_records; i++) {
        records[i].data = malloc(recsize + 1);
        fread(records[i].data, 1, recsize, in_cid);
        records[i].data[recsize] = '\0';
        records[i].key = extract_key(records[i].data, startbyte, numbytes);
    }
    fclose(in_cid);

    /* sort the records */
    qsort(records, 0, num_records-1);

    /* write the sorted records back to file */
```

```
        if ((in_cid = fopen(filename, "w")) == NULL) {
            printf("cound not open file %s for writing\n", in_cid);
            for (int i = 0; i < num_records; i++)
                free(records[i].data);
            free(records);
            return;
        }
        for (int i = 0; i < num_records; i++) {
            fwrite(records[i].data, 1, recsize, in_cid);
            free(records[i].data);
        }
        fclose(in_cid);
        free(records);
}
```

Debugging

> "cats miaow, dogs bark, politicians tell imaginative versions of the truth, and programmers make mistakes"

```
/* missing a semicolon */
p = p + x
q = q + y;

/* the compiler will see this code as */
p = p + xq = q + y;
```

This leads to an important observation: what the compiler sees as an error may not be the error you actually made.

Runtime Errors

> "If a C program is well-structured, it will consist of a number of short functions, each standing alone as a kind of mini-program (but perhaps callling other functions). And main will just put them all together in a clear way. So there's a sort of tree-like hierarchy of functions..."

> "There's a hoary old tale of a tramp sitting beside a heap of twigs striking matches"

```
/* power: return x power of y */
int power(int x, int y) {
    if (y == 0)
        return 1;

    int x0 = x;
```

```
    while (y > 1) {
        x *= x0;
        y--;
    }
    return x;
}

/* debug: print out dubug information */
void debug(char *message, int value) {
    char *message;
    int value;
    printf(">>%s %d", message, value);
}

/* debug_improved: that tracks down each times variable is called */
#define MAX_VARS 10
int _count[MAX_VARS] = {0}

void debug_improved(const char *message, int value, int refno) {
    _count[refno]++;

    printf(">>Ref %d: %s = %d: count = %d\n", refno, message,
            value, _count[refno]);
}
```

Rational Arithmetic

> "See a number as a pair of integers whose value is one of them divided by the other. Thus 3.2 appears as 32/10 or possibly 16/5. In general, let's talk about a number A as the pair (a,a') whose value is a/a'. It's obvious where the term rational arithmetic comes from; each number is a ratio of integers."

```
/* function to find the GCD using the Euclidean Algorithm */
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

/* The euclidean algorithm is based on the principle that the GCD of
 * two numbers also divides their difference.
 */
```

```c
/* Practical Organization   */
typedef struct {
    int top;
    int bottom;
} rat;

rat n, *p;
p = &n;
```