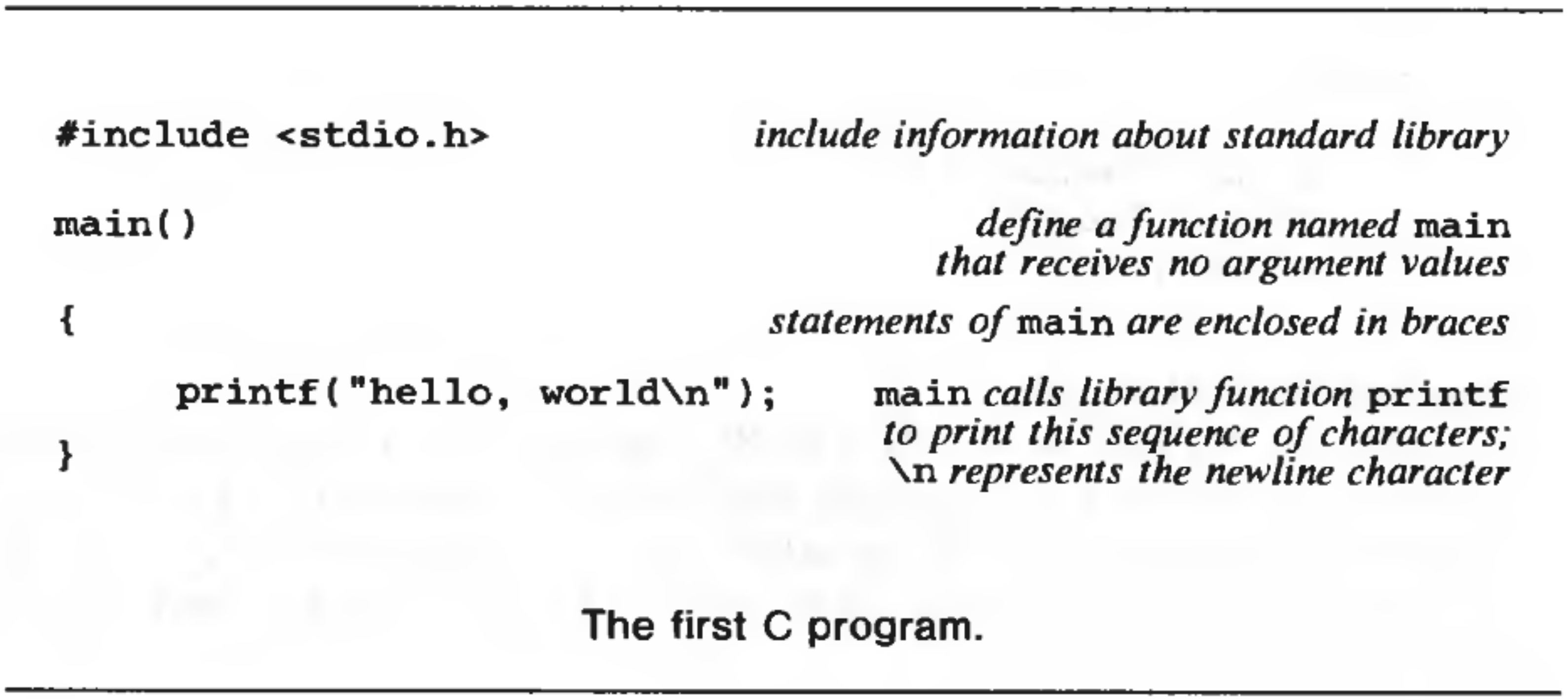
C



```
#include <stdio.h>                  include information about standard library

main()                                        define a function named main
                                          that receives no argument values

{                                statements of main are enclosed in braces

    printf("hello, world\n");         main calls library function printf
                                       to print this sequence of characters;
}                                       \n represents the newline character
```

The first C program.

Variables and Arithmetic Expressions

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
    for fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* lower limit of temperature table */
    upper = 300;    /* upper limit */
    step = 20;      /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32
        printf("%d\t%d\n", fah
        fahr = fahr + step;
    }
}
```

Comments

Declarations

Assignment statements

tab

`char`	character—a single byte
`short`	short integer
`long`	long integer
`double`	double-precision floating point

`%d`	print as decimal integer
`%6d`	print as decimal integer, at least 6 characters wide
`%f`	print as floating point
`%6f`	print as floating point, at least 6 characters wide
`%.2f`	print as floating point, 2 characters after decimal point
`%6.2f`	print as floating point, at least 6 wide and 2 after decimal point

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
    for fahr = 0, 20, ..., 300; floating-point version */
main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* lower limit of temperature table */
    upper = 300;    /* upper limit */
    step = 20;      /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Using for

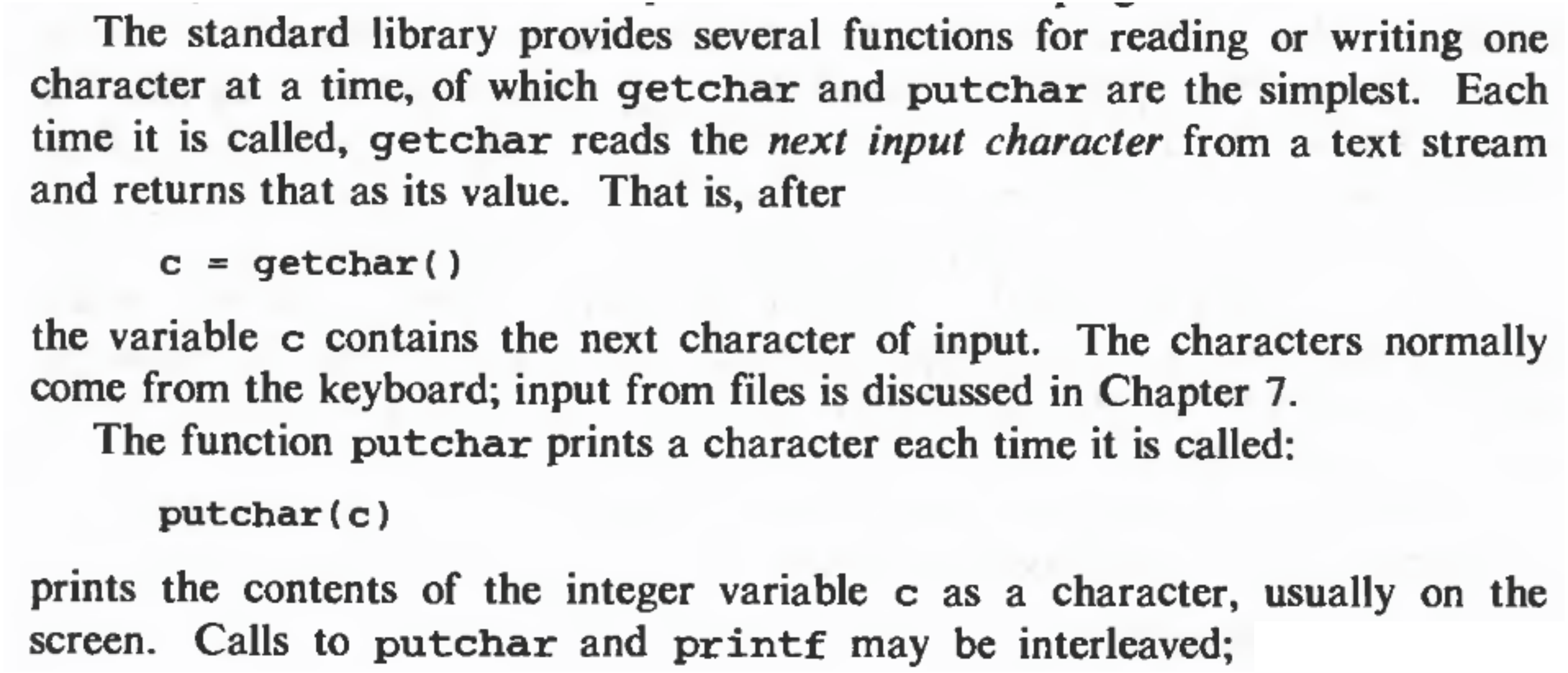
```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Symbolic constants

`#define` *name* *replacement text*

```
#include <stdio.h>

#define   LOWER  0       /* lower limit of table */
#define   UPPER  300     /* upper limit */
#define   STEP   20      /* step size */

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Character Input and Output



The standard library provides several functions for reading or writing one character at a time, of which `getchar` and `putchar` are the simplest. Each time it is called, `getchar` reads the *next input character* from a text stream and returns that as its value. That is, after

```
    c = getchar()
```

the variable `c` contains the next character of input. The characters normally come from the keyboard; input from files is discussed in Chapter 7.

The function `putchar` prints a character each time it is called:

```
    putchar(c)
```

prints the contents of the integer variable `c` as a character, usually on the screen. Calls to `putchar` and `printf` may be interleaved;

Getchar and putchar

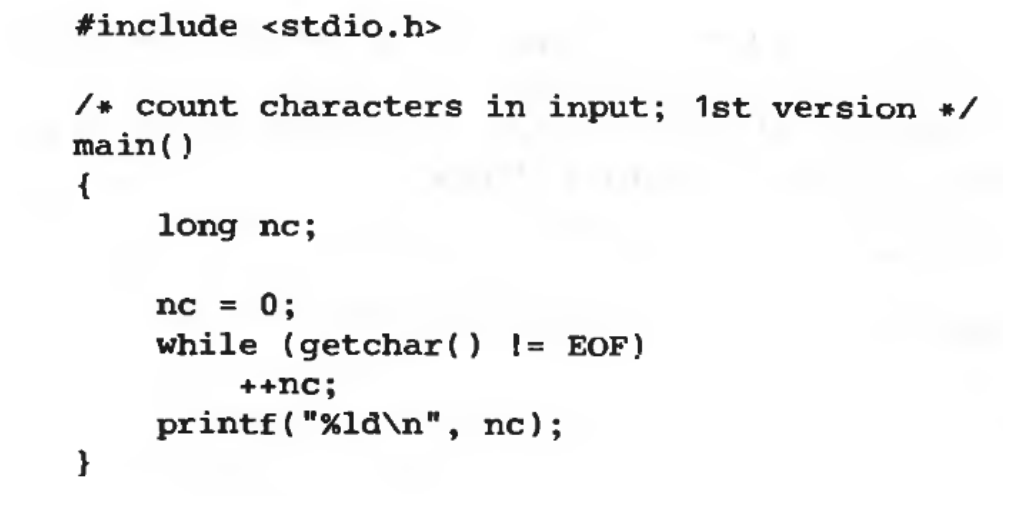
```
#include <stdio.h>

/* copy input to output; 1st version */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

```
#include <stdio.h>

/* copy input to output; 2nd version */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

The relational operator != means "not equal to."

The problem is distinguishing the end of the input from valid data. The solution is that getchar returns a distinctive value when there is no more input, a value that cannot be confused with any real character. This value is called EOF, for "end of file." We must declare c to be a type big enough to hold any value that getchar returns. We can't use char since c must be big enough to hold EOF in addition to any possible char. Therefore we use int.

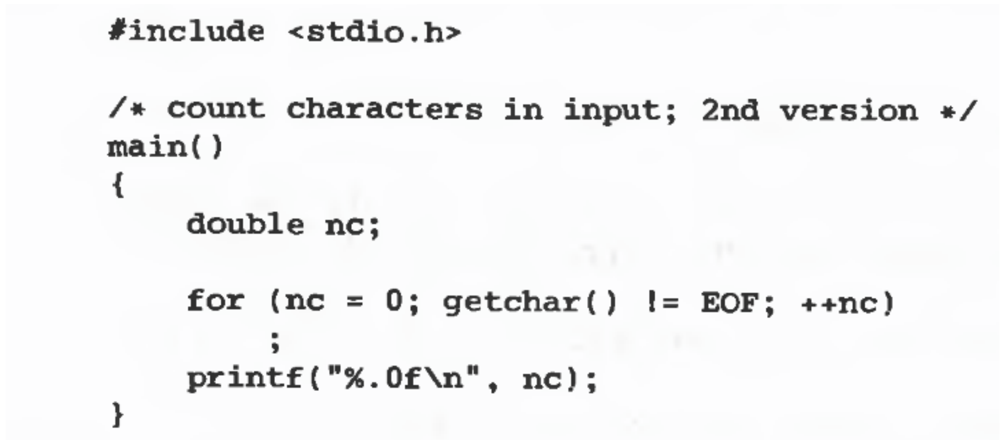
Character counting



```
#include <stdio.h>

/* count characters in input; 1st version */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Long = 32 bits (int = 16)

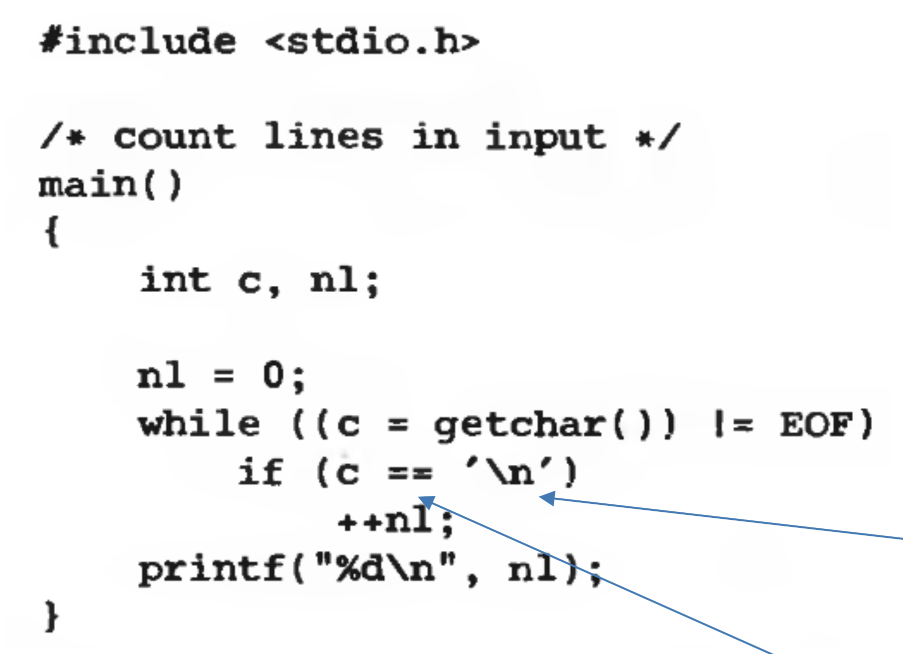
++ is increment by one (nc=nc+1) also nc++
n=5
X = n++ (x=5, n=6)
X = ++n (x=6, n=6)



```
#include <stdio.h>

/* count characters in input; 2nd version */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

Null statement

Line counting



```
#include <stdio.h>

/* count lines in input */
main()
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

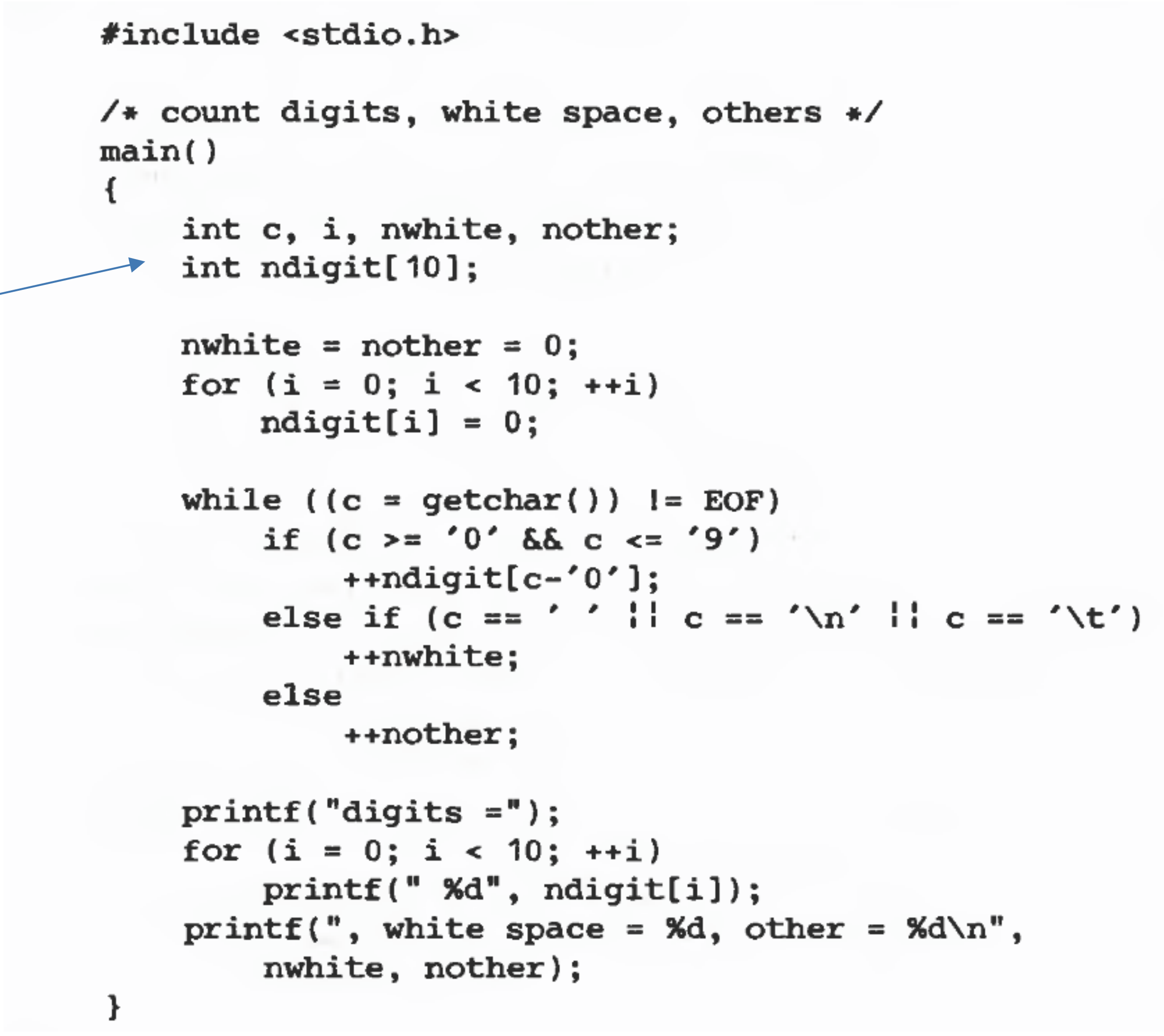
A character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set.

Same matlab or python

Arrays

Let us write a program to count the number of occurrences of each digit, of white space characters (blank, tab, newline), and of all other characters.

array



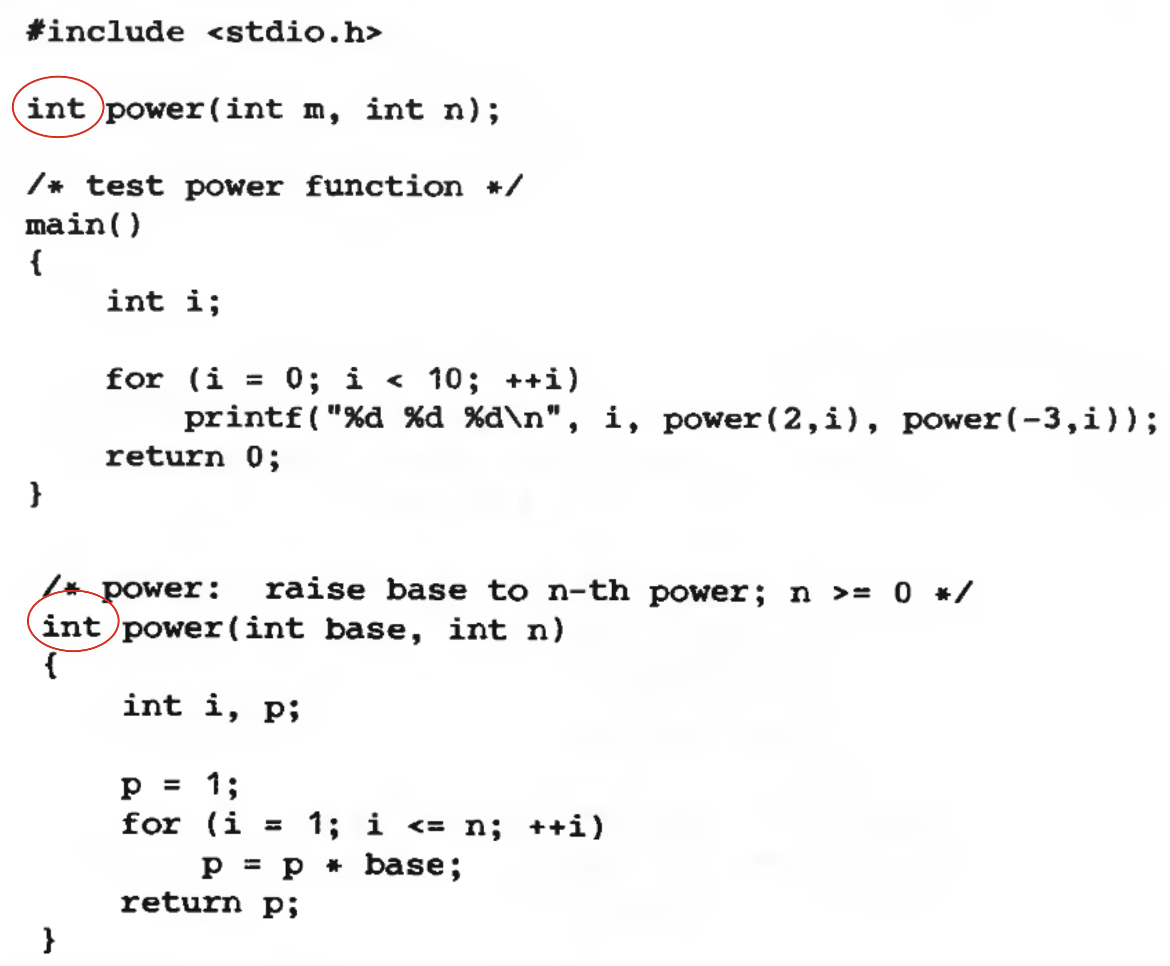
```c
#include <stdio.h>

/* count digits, white space, others */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
}
```

The output of this program on itself is

```
Digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

Functions

A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation.



```
#include <stdio.h>

int power(int m, int n);

/* test power function */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power:  raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

In C, all function arguments are passed "by value." This means that the called function is given the values of its arguments in temporary variables rather than the originals.

Character Arrays

No value returned

```c
#include <stdio.h>
#define MAXLINE 1000    /* maximum input line size */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print longest input line */
main()
{
    int len;              /* current line length */
    int max;              /* maximum length seen so far */
    char line[MAXLINE];     /* current input line */
    char longest[MAXLINE];  /* longest line saved here */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0)    /* there was a line */
        printf("%s", longest);
    return 0;
}
```

getline puts the character '\0' at the end of the array it is creating, to mark the end of the string of characters. This convention is also used by the C language:

h	e	l	l	o	\n	\0

```c
/* getline:  read a line into s, return length */
int getline(char s[], int lim)
{
    int c, i;

    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy:  copy 'from' into 'to'; assume to is big enough */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

External Variables and Scope

The variables in main, such as line, longest, etc., are private or local to main. Because they are declared within main, no other function can have direct access to them. As an alternative to automatic variables, it is possible to define variables that are external to all functions, that is, variables that can be accessed by name by any function

```
#include <stdio.h>

#define MAXLINE 1000    /* maximum input line size */

int max;                /* maximum length seen so far */
char line[MAXLINE];     /* current input line */
char longest[MAXLINE];  /* longest line saved here */

int getline(void);
void copy(void);



/* print longest input line; specialized version */
main()
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0)    /* there was a line */
        printf("%s", longest);
    return 0;
}
```

```
/* getline:  specialized version */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE-1
        && (c=getchar()) != EOF && c != '\n'; ++i)
            line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}
```

```
/* copy:  specialized version */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Punteros

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways.

The unary operator & gives the address of an object, so the statement

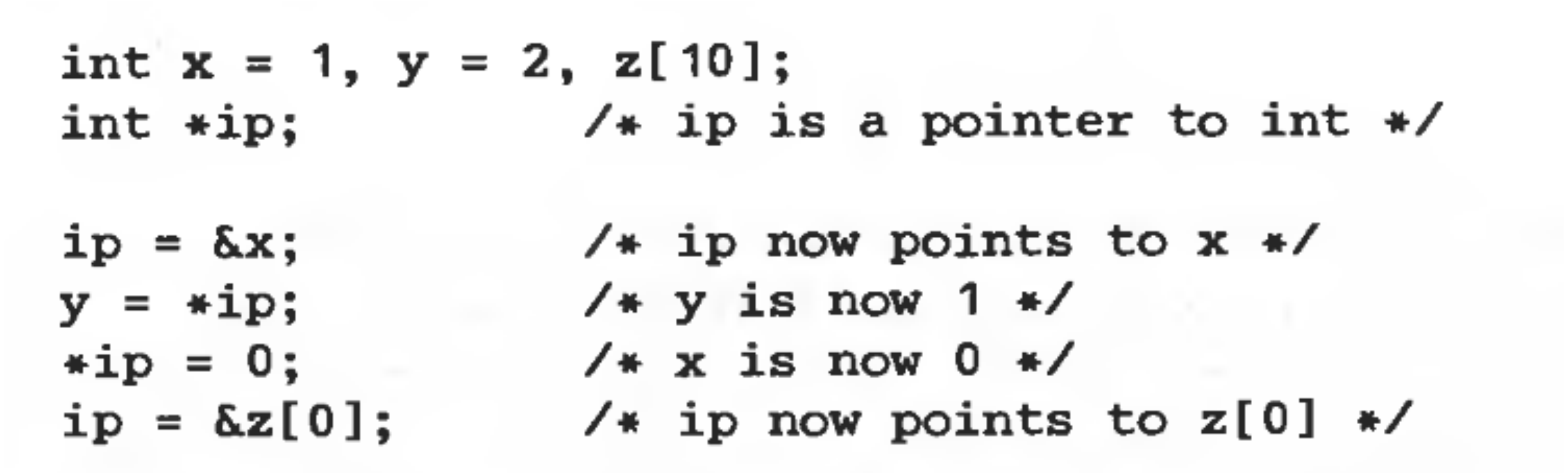
```
P = &c;
```

assigns the address of c to the variable p, and p is said to "point to" c.

The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

The unary operator * is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to.

Suppose that x and y are integers and ip is a pointer to int. This artificial sequence shows how to declare a pointer and how to use & and *:



```
int x = 1, y = 2, z[10];
int *ip;            /* ip is a pointer to int */

ip = &x;            /* ip now points to x */
y = *ip;            /* y is now 1 */
*ip = 0;            /* x is now 0 */
ip = &z[0];         /* ip now points to z[0] */
```

You should note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type.

If ip points to the integer x, then *ip can occur in any context where x could.

Since pointers are variables, they can be used without dereferencing.

For example, if iq is another pointer to int,

```
iq = ip
```

copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order elements with a function called swap.

It is not enough to write

swap(a, b);

where the swap function is defined as

```
void swap(int x, int y)  /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

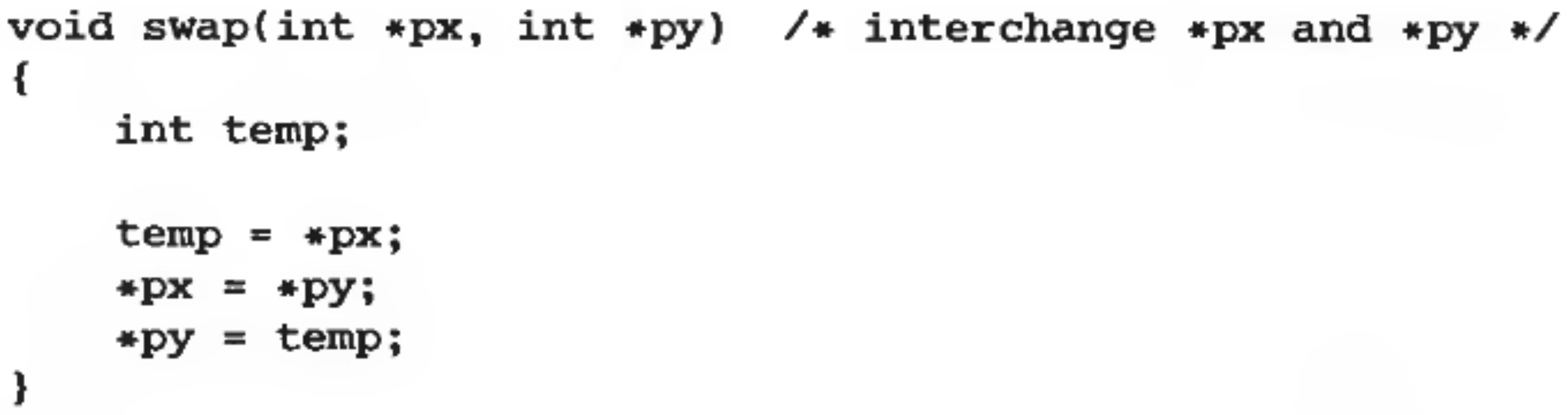
Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above only swaps copies of a and b.

The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

swap(&a, &b);

Since the operator & produces the address of a variable, &a is a pointer to a.

In swap itself, the parameters are declared to be pointers, and the operands are accessed indirectly through them.



```
void swap(int *px, int *py)  /* interchange *px and *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously.

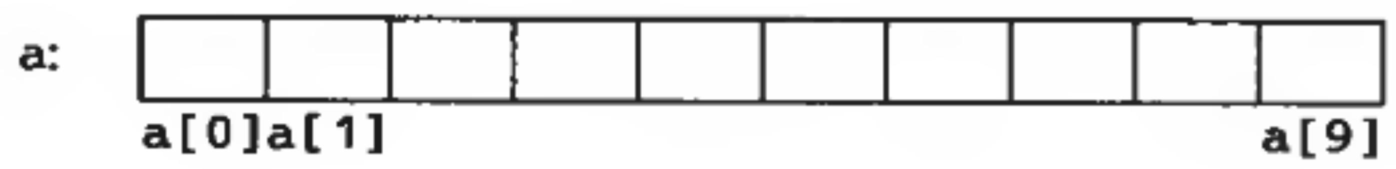
Any operation that can be achieved by array subscripting can also be done with pointers.

The pointer version will in general be faster but somewhat harder to understand.

The declaration

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

defines an array a of size 10, that is, a block of 10 consecutive objects named a[0], a[1], … a[9].



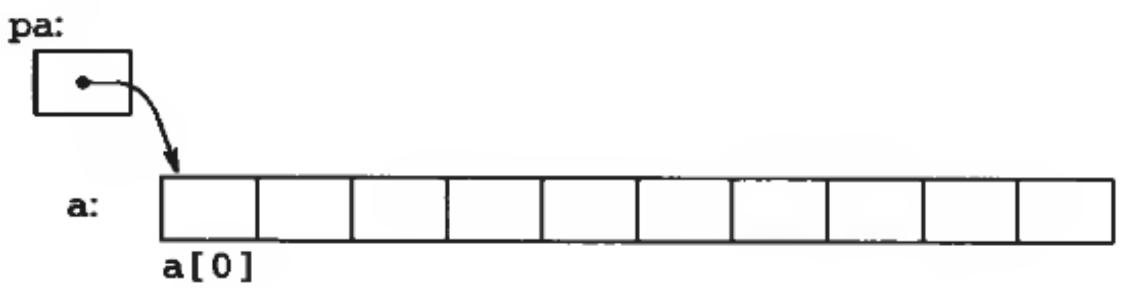
If pa is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets pa to point to element zero of a; that is, pa contains the address of a[0].



Now the assignment

```
x = *pa;
```

will copy the contents of a [0] into x.

If pa points to a particular element of an array, then by definition pa+1points to the next element,

pa+i points i elements after pa,

and

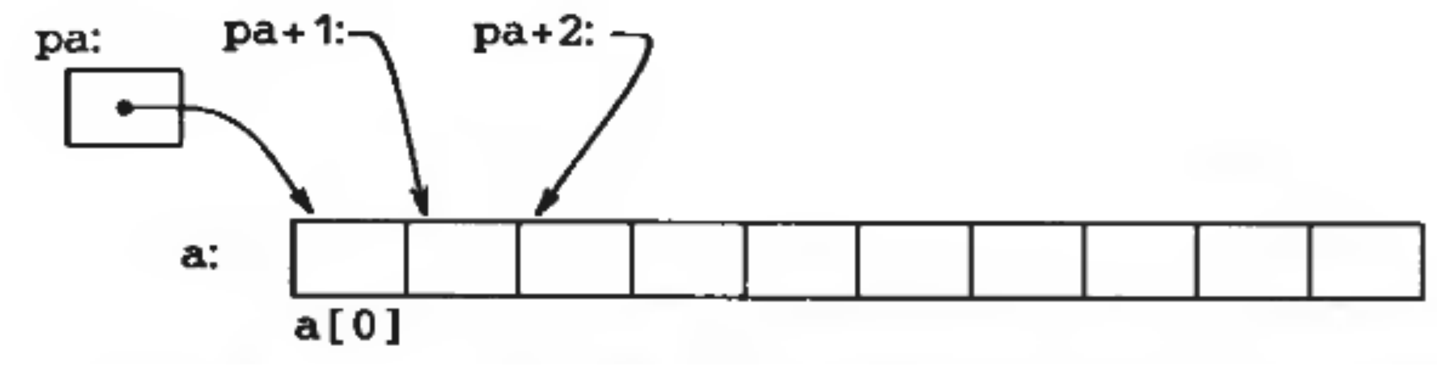
pa-i points ielements before.

Thus, if pa points to a[0],

pa+i is the address of a[i],

and

*(pa+i) is the contents of a[i].



These remarks are true regardless of the type or size of the variables in the array a.

The meaning of "adding 1 to a pointer," and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa.

The correspondence between indexing and pointer arithmetic is very close.

By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0];
```

pa and a have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment pa=&a [0] can also be writ-written as

```
pa = a;
```

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so pa=a and pa++ are legal. But an array name is not a variable; constructions like a=pa and a++ are illegal.