

and the sequence

```
y = &x[0];  
y++;
```

leaves `y` pointing at `x[1]`.

Let's use pointers in a function `length` that computes how long a character array is. Remember that by convention all character arrays are terminated with a `'\0'`. (And if they aren't, this program will blow up inevitably.) The old way:

```
length(s)  
    char s[ ]; {  
        int n;  
        for( n=0; s[n] != '\0'; )  
            n++;  
        return(n);  
    }
```

Rewriting with pointers gives

```
length(s)  
    char *s; {  
        int n;  
        for( n=0; *s != '\0'; s++ )  
            n++;  
        return(n);    }
```

You can now see why we have to say what kind of thing `s` points to -- if we're to increment it with `s++` we have to increment it by the right amount.

The pointer version is more efficient (this is almost always true) but even more compact is

```
for( n=0; *s++ != '\0'; n++ );
```

The `'*s'` returns a character; the `'++'` increments the pointer so we'll get the next character next time around. As you can see, as we make things more efficient, we also make them less clear. But `'*s++'` is an idiom so common that you have to know it.

Going a step further, here's our function `strcpy` that copies a character array `s` to another `t`.

```
strcpy(s,t)  
    char *s, *t; {  
        while(*t++ = *s++);  
    }
```

We have omitted the test against `'\0'`, because `'\0'` is identically zero; you will often see the code this way.

For arguments to a function, and there only, the declarations

```
char s[ ];  
char *s;
```

are equivalent -- a pointer to a type, or an array of unspecified size of that type, are the same thing.

## Function Arguments

Look back at the [function `strcpy` in the previous section](#). We passed it two string names as arguments, then proceeded to clobber both of them by incrementation. So how come we don't lose the original strings in the function that called `strcpy`?

As we said before, C is a "call by value" language: when you make a function call like `f(x)`, the *value* of `x` is passed, not its address. So there's no way to *alter* `x` from inside `f`. If `x` is an array (`char x[10]`) this isn't a problem, because `x` *is* an address anyway, and you're not trying to change it, just what it addresses. This is why `strcpy` works as it does. And it's convenient not to have to worry about making temporary copies of the input arguments.

But what if `x` is a scalar and you do want to change it? In that case, you have to pass the *address* of `x` to `f`, and then use it as a pointer. Thus for example, to interchange two integers, we must write

```
flip(x, y)  
{  
    int *x, *y; {  
        int temp;  
        temp = *x;  
        *x = *y;  
        *y = temp;  
    }  
}
```

and to call `flip`, we have to pass the addresses of the variables:

```
flip (&a, &b);
```

Which interchange two integers.

## The Switch Statement ; Break ; Continue

The switch statement can be used to replace the multi-way test we used in the last example. When the tests are like this:

```
if( c == 'a' ) ...
else if( c == 'b' ) ...
else if( c == 'c' ) ...
else ...
```

testing a value against a series of constants, the switch statement is often clearer and usually gives better code. Use it like this:

```
switch( c ) {

case 'a':
    aflag++;
    break;
case 'b':
    bflag++;
    break;
case 'c':
    cflag++;
    break;
default:
    printf("%c?\n", c);
    break;
}
```

The case statements label the various actions we want; `default` gets done if none of the other cases are satisfied. (A `default` is optional; if it isn't there, and none of the cases match, you just fall out the bottom.)

The `break` statement in this example is new. It is there because the cases are just labels, and after you do one of them, you fall through to the next unless you take some explicit action to escape. This is a mixed blessing. On the positive side, you can have multiple cases on a single statement; we might want to allow both upper and lower

But what if we just want to get out after doing case 'a' ? We could get out of a `case` of the `switch` with a `goto`, but this is really ugly. The `break` statement lets us exit without either `goto` or `label`.

The `break` statement also works in `for` and `while` statements; it causes an immediate exit from the loop.

The `continue` statement works *only* inside `for`'s and `while`'s; it causes the next iteration of the loop to be started. This means it goes to the increment part of the `for` and the test part of the `while`.

## Structures

The main use of structures is to lump together collections of disparate variable types, so they can conveniently be treated as a unit. For example, if we were writing a compiler or assembler, we might need for each identifier information like its name (a character array), its source line number (an integer), some type information (a character, perhaps), and probably a usage count (another integer).

```
char    id[10];
int     line;
char    type;
int     usage;
```

We can make a structure out of this quite easily. We first tell C what the structure will look like, that is, what kinds of things it contains; after that we can actually reserve storage for it, either in the same statement or separately. The simplest thing is to define it and allocate storage all at once:

```
struct {
    char    id[10];
    int     line;
    char    type;
    int     usage;      } sym;
```

This defines `sym` to be a structure with the specified shape; `id`, `line`, `type` and `usage` are members of the structure. The way we refer to any particular member of the structure is

```
structure-name . member
```

as in

```
sym.type = 077;
if( sym.usage == 0 ) ...
while( sym.id[j++] ) ...
etc.
```

Although the names of structure members never stand alone, they still have to be unique; there can't be another `id` or `usage` in some other structure.

So far we haven't gained much. The advantages of structures start to come when we have arrays of structures, or when we want to pass complicated data layouts between functions. Suppose we wanted to make a symbol table for up to 100 identifiers. We could extend our definitions like

```
char    id[100][10];
int     line[100];
char    type[100];
int     usage[100];
```

but a structure lets us rearrange this spread-out information so all the data about a single identifier is collected into one lump:

```
struct {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100];
```

This makes `sym` an array of structures; each array element has the specified shape. Now we can refer to members as

```
sym[i].usage++; /* increment usage of i-th identifier */
for( j=0; sym[i].id[j++] != '\0'; ) ...
etc.
```

Thus to print a list of all identifiers that haven't been used, together with their line number,

```
for( i=0; i<nsym; i++ )
    if( sym[i].usage == 0 )
        printf("%d\t%s\n", sym[i].line, sym[i].id);
```

Suppose we now want to write a function `lookup(name)` which will tell us if `name` already exists in `sym`, by giving its index, or that it doesn't, by returning a -1. We can't pass a structure to a function directly; we have to either define it externally, or pass a pointer to it. Let's try the first way first.

```
int     nsym    0;      /* current length of symbol table */

struct {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100];             /* symbol table */
```

```
main( ) {
    ...
    if( (index = lookup(newname)) >= 0 )
        sym[index].usage++;      /* already there ... */
    else
        install(newname, newline, newtype);
    ...
}

lookup(s)
char *s; {
    int i;
    extern struct {
        char    id[10];
        int     line;
        char    type;
        int     usage;
    } sym[ ];

    for( i=0; i<nsym; i++ )
        if( compar(s, sym[i].id) > 0 )
            return(i);

    return(-1);
}

compar(s1,s2)      /* return 1 if s1==s2, 0 otherwise */
char *s1, *s2; {
    while( *s1++ == *s2 )
        if( *s2++ == '\0' )
            return(1);

    return(0);
}
```

The declaration of the structure in `lookup` isn't needed if the external definition precedes its use in the same source file, as we shall see in a moment.

Now what if we want to use pointers?

```
struct symtag {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100], *psym;

psym = &sym[0]; /* or p = sym; */
```

This makes `psym` a pointer to our kind of structure (the symbol table), then initializes it to point to the first element of `sym`.

Notice that we added something after the word `struct`: a ```tag"` called `symtag`. This puts a name on our structure definition so we can refer to it later without repeating the definition. It's not necessary but useful. In fact we could have said

```
struct  symtag {
    ... structure definition
};
```

which wouldn't have assigned any storage at all, and then said

```
struct  symtag  sym[100];
struct  symtag  *psym;
```

which would define the array and the pointer. This could be condensed further, to

```
struct  symtag  sym[100], *psym;
```

The way we actually refer to an member of a structure by a pointer is like this:

```
ptr -> structure-member
```

The symbol ``->'` means we're pointing at a member of a structure; ``->'` is only used in that context. `ptr` is a pointer to the (base of) a structure that contains the structure member. The expression `ptr->structure-member` refers to the indicated member of the pointed-to structure. Thus we have constructions like:

```
psym->type = 1;
psym->id[0] = 'a';
```

For more complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example,

```
struct { int x, *y; } *p;
p->x++ increments x
++p->x so does this!
(++p)->x increments p before getting x
*p->y++ uses y as a pointer, then increments it
*(p->y)++ so does this
*(p++)->y uses y as a pointer, then increments p
```

The way to remember these is that `->`, `.` (dot), `( )` and `[ ]` bind very tightly. An expression involving one of these is treated as a unit. `p->x`, `a[i]`, `y.x` and `f(b)` are names exactly as `abc` is.

If `p` is a pointer to a structure, any arithmetic on `p` takes into account the actual size of the structure. For instance, `p++` increments `p` by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes

of its members -- because of alignments of different sized objects, there may be "holes" in a structure.

Enough theory. Here is the lookup example, this time with pointers.

```

struct symtag {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100];

main( ) {
    struct symtag *lookup( );
    struct symtag *psym;
    ...
    if( (psym = lookup(newname)) ) /* non-zero pointer */
        psym -> usage++;           /* means already there */
    else
        install(newname, newline, newtype);
    ...
}

struct symtag *lookup(s)
    char *s; {
    struct symtag *p;
    for( p=sym; p < &sym[nsym]; p++ )
        if( compar(s, p->id) > 0)
            return(p);
    return(0);
}

```

The function `compar` doesn't change: `'p->id'` refers to a string.

In `main` we test the pointer returned by `lookup` against zero, relying on the fact that a pointer is by definition never zero when it really points at something. The other pointer manipulations are trivial.

The only complexity is the set of lines like

```

struct symtag *lookup( );

```

This brings us to an area that we will treat only hurriedly; the question of function types. So far, all of our functions have returned integers (or characters, which are much the same). What do we do when the function returns something else, like a pointer to a structure? The rule is that any function that doesn't return an `int` has to say explicitly what it does return. The type information goes before the function name (which can make the name hard to see).



Examples:

```
char f(a)
    int a; {
        ...
    }
int *g( ) { ... }

struct symtag *lookup(s) char *s; { ... }
```

The function `f` returns a character, `g` returns a pointer to an integer, and `lookup` returns a pointer to a structure that looks like `symtag`. And if we're going to use one of these functions, we have to make a declaration where we use it, as we did in `main` above.

Notice the parallelism between the declarations

```
struct symtag *lookup( );
struct symtag *psym;
```

In effect, this says that `lookup( )` and `psym` are both used the same way - as a pointer to a structure -- even though one is a variable and the other is a function.

## Initialization of Variables

An external variable may be initialized at compile time by following its name with an initializing value when it is defined. The initializing value has to be something whose value is known at compile time, like a constant.

```
int    x = 0;      /* "0" could be any constant */
int    *p    &y[1]; /* p now points to y[1] */
```

An external array can be initialized by following its name with a list of initializations enclosed in braces:

```
int    x[4] = {0,1,2,3}; /* makes x[i] = i */
int    y[ ] = {0,1,2,3}; /* makes y big enough for 4 values */
char    *msg = "syntax error\n"; /* braces unnecessary here */
char    *keyword[ ]={
        "if",
        "else",
        "for",
        "while",
        "break",
        "continue",
        0
    };
```

This last one is very useful -- it makes keyword an array of pointers to character strings, with a zero at the end so we can identify the last element easily. A simple lookup routine could scan this until it either finds a match or encounters a zero keyword pointer:

```
lookup(str)          /* search for str in keyword[ ] */
char *str; {
    int i,j,r;
    for( i=0; keyword[i] != 0; i++) {
        for( j=0; (r=keyword[i][j]) == str[j] && r != '\0'; j++)
    );
        if( r == str[j] )
            return(i);
    }
    return(-1);
}
```

## Scope Rules

A complete C program need not be compiled all at once; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. How do we arrange that data gets passed from one routine to another? We have already seen how to use function arguments and values, so let us talk about external data. Warning: the words declaration and definition are used precisely in this section; don't treat them as the same thing.

A major shortcut exists for making extern declarations. If the definition of a variable appears *before* its use in some function, no extern declaration is needed within the function. Thus, if a file contains

```
f1( ) { ... }
int foo;
f2( ) { ... foo = 1; ... }
f3( ) { ... if ( foo ) ... }
```

no declaration of `foo` is needed in either `f2` or `f3`, because the external definition of `foo` appears before them. But if `f1` wants to use `foo`, it has to contain the declaration

```
f1( ) {
    extern int foo;
    ...
}
```

This is true also of any function that exists on another file; if it wants `foo` it has to use an extern declaration for it. (If somewhere there is an extern declaration for something, there must also eventually be an external definition of it, or you'll get an "undefined symbol" message.)

There are some hidden pitfalls in external declarations and definitions if you use multiple source files. To avoid them, first, define and initialize each external variable only once in the entire set of files:

```
int    foo = 0;
```

You can get away with multiple external definitions on UNIX, but not on GCOS, so don't ask for trouble. Multiple initializations are illegal everywhere. Second, at the beginning of any file that contains functions needing a variable whose definition is in some other file, put in an extern declaration, outside of any function:

```
extern int    foo;

fl( ) { ... }      etc.
```

## #define, #include

C provides a very limited macro facility. You can say

```
#define name          something
```

and thereafter anywhere ``name" appears as a token, ``something" will be substituted. This is particularly useful in parametering the sizes of arrays:

```
#define ARRAYSIZE      100
int    arr[ARRAYSIZE];
...
while( i++ < ARRAYSIZE )...
```

(now we can alter the entire program by changing only the define) or in setting up mysterious constants:

```
#define SET            01
#define INTERRUPT      02    /* interrupt bit */
#define ENABLED 04
...
if( x & (SET | INTERRUPT | ENABLED) ) ...
```

Now we have meaningful words instead of mysterious constants. (The mysterious operators '&' (AND) and '|' (OR) will be covered in the [next section](#).) It's an excellent practice to write programs without any literal constants except in #define statements.

There are several warnings about #define. First, there's no semicolon at the end of a #define; all the text from the name to the end of the line (except for comments) is taken to be the ``something". When it's put into the text, blanks are placed around it. The other

control word known to C is `#include`. To include one file in your source at compilation time, say

```
#include "filename"
```

### Bit Operators

C has several operators for logical bit-operations. For example,

```
x = x & 0177;
```

forms the bit-wise AND of `x` and 0177, effectively retaining only the last seven bits of `x`. Other operators are

	<i>inclusive OR</i>
^	<i>(circumflex) exclusive OR</i>
~	<i>(tilde) 1's complement</i>
!	<i>logical NOT</i>
<<	<i>left shift (as in x&lt;&lt;2)</i>
>>	<i>right shift (arithmetic on PDP-11; logical on H6070,</i>
	<i>IBM360)</i>

### Assignment Operators

An unusual feature of C is that the normal binary operators like `+`, `-`, etc. can be combined with the assignment operator `=` to form new assignment operators. For example,

```
x -= 10;
```

uses the assignment operator `-=` to decrement `x` by 10, and

```
x =& 0177
```

forms the AND of `x` and 0177. This convention is a useful notational shortcut, particularly if `x` is a complicated expression. The classic example is summing an array:

```
for( sum=i=0; i<n; i++ )
    sum += array[i];
```

But the spaces around the operator are critical! For

```
x = -10;
```

also decreases `x` by 10. This is quite contrary to the experience of most programmers. In particular, watch out for things like

```
c=*s++;
y=&x[0];
```

both of which are almost certainly not what you wanted. Newer versions of various compilers are courteous enough to warn you about the ambiguity.

Because all other operators in an expression are evaluated before the assignment operator, the order of evaluation should be watched carefully:

```
x = x<<y | z;
```

means ``shift x left y places, then OR with z, and store in x."

## Floating Point

C has single and double precision numbers. For example,

```
double sum;
float avg, y[10];
sum = 0.0;
for( i=0; i<n; i++ )
    sum += y[i]; avg = sum/n;
```

All floating arithmetic is done in double precision. Mixed mode arithmetic is legal; if an arithmetic operator in an expression has both operands `int` or `char`, the arithmetic done is integer, but if one operand is `int` or `char` and the other is `float` or `double`, both operands are converted to `double`. Thus if `i` and `j` are `int` and `x` is `float`,

<code>(x+i)/j</code>	<i>converts i and j to float</i>
<code>x + i/j</code>	<i>does i/j integer, then converts</i>

Type conversion may be made by assignment; for instance,

```
int m, n;
float x, y;
m = x;
y = n;
```

converts `x` to integer (truncating toward zero), and `n` to floating point.

Floating constants are just like those in Fortran or PL/I, except that the exponent letter is `'e'` instead of `'E'`. Thus:

```
pi = 3.14159;
large = 1.23456789e10;
```

`printf` will format floating point numbers: ```%w.df"` in the format string will print the corresponding variable in a field `w` digits wide, with `d` decimal places. An `e` instead of an `f` will produce exponential notation.

### goto and labels

C has a `goto` statement and labels, so you can branch about the way you used to. But most of the time `goto`'s aren't needed. (How many have we used up to this point?) The code can almost always be more clearly expressed by `for/while`, `if/else`, and compound statements.

One use of `goto`'s with some legitimacy is in a program which contains a long loop, where a `while(1)` would be too extended. Then you might write

```
mainloop:
    ...
    goto mainloop;
```

Another use is to implement a `break` out of more than one level of `for` or `while`. `goto`'s can only branch to labels within the same function.

### Manipulating Strings

A string is a sequence of characters, with its beginning indicated by a pointer and its end marked by the null character `\0`. At times, you need to know the length of a string (the number of characters between the start and the end of the string) [5]. This length is obtained with the library function `strlen()`. Its prototype, in `STRING.H`, is

```
size_t strlen(char *str);
```

### The strcpy() Function

The library function `strcpy()` copies an entire string to another memory location. Its prototype is as follows:

```
char *strcpy( char *destination, char *source );
```

**Before using `strcpy()`, you must allocate storage space for the destination string.**

```
/* Demonstrates strcpy(). */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char source[] = "The source string.";
```

```
main()
{
    char dest1[80];
    char *dest2, *dest3;

    printf("\nsource: %s", source );

    /* Copy to dest1 is okay because dest1 points to */
    /* 80 bytes of allocated space. */

    strcpy(dest1, source);
    printf("\ndest1: %s", dest1);

    /* To copy to dest2 you must allocate space. */
    dest2 = (char *)malloc(strlen(source) +1);
    strcpy(dest2, source);
    printf("\ndest2: %s\n", dest2);

    return(0);
}
source: The source string.
dest1:  The source string.
dest2:  The source string.
```

### The strncpy() Function

The `strncpy()` function is similar to `strcpy()`, except that `strncpy()` lets you specify how many characters to copy. Its prototype is

```
char *strncpy(char *destination, char *source, size_t n);
```

```
/* Using the strncpy() function. */

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

char dest[] = ".....";
char source[] = "abcdefghijklmnopqrstuvwxyz";

main()
{
    size_t n;

    while (1)
    {
        puts("Enter the number of characters to copy (1-26)");
        scanf("%d", &n);

        if (n > 0 && n< 27)
            break;
    }
}
```

```
printf("\nBefore strncpy destination = %s", dest);

strncpy(dest, source, n);

printf("\nAfter strncpy destination = %s\n", dest);
return(0); }
```

Enter the number of characters to copy (1-26)

**15**

Before strncpy destination = .....

After strncpy destination = abcdefghijklmno.....

### The strdup() Function

The library function `strdup()` is similar to `strcpy()`, except that `strdup()` performs its own memory allocation for the destination string with a call to `malloc()`. The prototype for `strdup()` is

```
char *strdup( char *source );
```

**Using `strdup()` to copy a string with automatic memory allocation.**

```
/* The strdup() function. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char source[] = "The source string.";

main()
{
    char *dest;

    if ( (dest = strdup(source)) == NULL)
    {
        fprintf(stderr, "Error allocating memory.");
        exit(1);
    }

    printf("The destination = %s\n", dest);
    return(0);
}
The destination = The source string.
```



## The strcat() Function

The prototype of `strcat()` is

```
char *strcat(char *str1, char *str2);
```

The function appends a copy of `str2` onto the end of `str1`, moving the terminating null character to the end of the new string. You must allocate enough space for `str1` to hold the resulting string. The return value of `strcat()` is a pointer to `str1`. Following listing demonstrates `strcat()`.

```
/* The strcat() function. */

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

char str1[27] = "a";
char str2[2];

main()
{
    int n;

    /* Put a null character at the end of str2[]. */

    str2[1] = '\0';

    for (n = 98; n < 123; n++)
    {
        str2[0] = n;
        strcat(str1, str2);
        puts(str1);
    }
    return(0);
}
```

```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
```

```
abcdefghijklmnop  
abcdefghijklmnopq  
abcdefghijklmnopqr  
abcdefghijklmnopqrs  
abcdefghijklmnopqrst  
abcdefghijklmnopqrstu  
abcdefghijklmnopqrstuv  
abcdefghijklmnopqrstuvw  
abcdefghijklmnopqrstuvwx  
abcdefghijklmnopqrstuvwxy  
abcdefghijklmnopqrstuvwxyz
```

## Comparing Strings

Strings are compared to determine whether they are equal or unequal. If they are unequal, one string is "greater than" or "less than" the other. Determinations of "greater" and "less" are made with the ASCII codes of the characters. In the case of letters, this is equivalent to alphabetical order, with the one seemingly strange exception that all uppercase letters are "less than" the lowercase letters. This is true because the uppercase letters have ASCII codes 65 through 90 for A through Z, while lowercase a through z are represented by 97 through 122. Thus, "ZEBRA" would be considered to be less than "apple" by these C functions.

The ANSI C library contains functions for two types of string comparisons: comparing two entire strings, and comparing a certain number of characters in two strings.

## Comparing Two Entire Strings

The function `strcmp()` compares two strings character by character. Its prototype is

```
int strcmp(char *str1, char *str2);
```

The arguments `str1` and `str2` are pointers to the strings being compared. The function's return values are given in Table. Following Listing demonstrates `strcmp()`.

**The values returned by `strcmp()`.**

Return Value	Meaning
< 0	str1 is less than str2.
0	str1 is equal to str2.
> 0	str1 is greater than str2.

**Using strcmp() to compare strings.**

```
/* The strcmp() function. */

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

main()
{
    char str1[80], str2[80];
    int x;

    while (1)
    {

        /* Input two strings. */
        printf("\n\nInput the first string, a blank to exit: ");
        gets(str1);

        if ( strlen(str1) == 0 )
            break;

        printf("\nInput the second string: ");
        gets(str2);

        /* Compare them and display the result. */

        x = strcmp(str1, str2);

        printf("\nstrcmp(%s,%s) returns %d", str1, str2, x);
    }
    return(0);
}
```

```
Input the first string, a blank to exit: First string
Input the second string: Second string
strcmp(First string,Second string) returns -1
Input the first string, a blank to exit: test string
Input the second string: test string
strcmp(test string,test string) returns 0
Input the first string, a blank to exit: zebra
Input the second string: aardvark
strcmp(zebra,aardvark) returns 1
Input the first string, a blank to exit:
```

**Comparing Partial Strings**

The library function strncmp() compares a specified number of characters of one string to another string. Its prototype is

```
int strncmp(char *str1, char *str2, size_t n);
```

The function `strncmp()` compares `n` characters of `str2` to `str1`. The comparison proceeds until `n` characters have been compared or the end of `str1` has been reached. The method of comparison and return values are the same as for `strcmp()`. The comparison is case-sensitive.

### Comparing parts of strings with `strncmp()`.

```
/* The strncmp() function. */

#include <stdio.h>
#include [Sigma]>tring.h>

char str1[] = "The first string.";
char str2[] = "The second string.";
main()
{
    size_t n, x;

    puts(str1);
    puts(str2);

    while (1)
    {
        puts("\n\nEnter number of characters to compare, 0 to exit.");
        scanf("%d", &n);

        if (n <= 0)
            break;

        x = strncmp(str1, str2, n);

        printf("\nComparing %d characters, strncmp() returns %d.", n, x);
    }
    return(0);
}
```

```
The first string.
The second string.
Enter number of characters to compare, 0 to exit.
3
Comparing 3 characters, strncmp() returns .@]
Enter number of characters to compare, 0 to exit.
6
Comparing 6 characters, strncmp() returns -1.
Enter number of characters to compare, 0 to exit.
0
```