Let us examine four ways to copy null-terminated strings:
s (for **source**) is a pointer to a string,
t (for **target**) pointer to at least 1+strlen(s) bytes of memory.

```
   /* Array version */
void Astrcopy ( char s[ ], char t[ ] ) {
    int i = 0;
    while ( (t[i] = s[i]) != '\0' )
        i++;
}
    /* Pointer version 1 */
void P1strcopy ( char* s, char* t ) {
    int i = 0;
    while ((*(t+i) = *(s+i)) != '\0')
        i++;
}
    /* Pointer version 2 */
void P2strcopy ( char* s, char* t ) {
    while ((*t++ = *s++) != '\0')
        ;
}
    /* Pointer version 3 */
void P3strcopy ( char* s, char* t ) {
    while (*t++ = *s++)
        ;
}
```


Pointers to Functions

Use of a pointer to a function provides an elegant solution to some problems.

A pointer to a function holds the memory address of the function's executable cod
Using (dereferencing) the pointer is the same as calling the function.

Function pointers can only be used in the following way:

- **assignment to** (specify function name)
- **assignment from** (make a copy)
- **dereference** (that is, make function call)

Consider the prototype for some function (here display):

```
int display ( int, char* );
```

we can declare the existence of a function pointer **fptr**:

```
int (*fptr)();
```

The general form of the syntax is:

```
type (*pointer)();
```


Initialize the pointer by simply assigning the name to it.

```
fptr = display;
```

We can now invoke (call) this function via the pointer

```
value = (*fptr) ( 64, "Industrial Internship wins" );
```

But what are they really good for?

1. Passing a function as a parameter
   By this means different operations can be done with the same piece of code

2. To select a function to call at run-time, so avoiding ugly (or impractical) switch statement.

For example, we want to read a key press and call a different action (function) depending on the key selected. For this we define an array of function pointers as follows:


```
int (*RespondToKeypress[256])();

RespondToKeypress['\t'] = tab_handler;

for ( i = 'a'; i <= 'z'; i++)
   RespondToKeypress[i] = alpha_handler;
```

So now when some key c is pressed we invoke

```
RespondToKeypress[c](c);
```

and the job is done

Similarly to a pass function as a parameter:

- Want some action when selecting a graphical object.
  e.g., Please call **this function** when the user presses **this button**

- Functions as objects. As we will see later, C++ classes are essentially structures which include "member functions" (operations on the object). In C much of the equivalent can be achieved using pointers to functions.

- System calls to sort or search, where you need to tell the system how to compare data types

All sorting functions have at their heart an evaluation function that is used to rank two elements.

The algorithm is independent of the details of this comparison, so why not write a generic sort function and pass the comparison routine as a parameter?

Consider the sorting of an array of chars using qsort

```
void qsort (void* base, int nel, int width, int (*CPtr)(void*, void*));
```

base	The origin of the array of elements
nel	The number of elements in the array
width	The size of each element (bytes)
CPtr	pointer to a function whose two parameters are pointers to two elements. CPtr returns -1, 0, +1

For sorting "chars" we might define a compare function:

```
int CmpChar ( char* a, char* b) {
   if ( *a == *b ) {
      return 0;
   } else {
      return ( *a > *b ? 1 : -1 );
   }
}

int (*CmpPtr)();          // declare a function pointer
CmpPtr = CmpChar;         // initialize it

char buffer[32] = {'+', '9', '"', 'w', 'e', 'R'};

qsort ( buffer, 32, sizeof(char), CmpPtr );
```

See also Unix quicksort program, King 173-176.

Regular Expressions (for pattern matching)

strcmp (char*, char*) can be used for exact matching of strings, but often we want to find substrings that conform to certain constraints.

Regular expressions specify a pattern to be sought in an array of characters. We can define our regular expression and then ask the system if particular strings conform to it.

Regular expressions are found throughout the UNIX environment, both at the command line (through **grep**--Get Regular ExPression), and through system calls from within user programs.

A few single character-matching codes

^ denotes the start of a line

$ denotes the end of the line

c Any character; use **\c** if special (like, **^**, **&**, **.**, ****, **%**, **/** etc.)

. any single character

[abc] a, b or c are matched

[d-k] any of d, e, f, g, h, i, j or k are matched

[^xyz] any character not in the set {xyz}

Operators

***** Will **match n duplications** of an expression (n can be zero, NULL string)
xy expression x followed by expression y

\(...\) delimits a subexpression
\n matches the n-th subexpression found in this line, n in [0,9].

UNIX grep command

grep 'regex' ListofFiles

scans all the characters in the files specified, and prints out every line that contains at least one match with the given regular expression ‘regex’.
e.g.

```
cd /usr/include
grep '.printf' *.h
```

scans all the .h files and prints out lines containing .printf
Note you need the single quotes here, since the regular expression contains the special symbol [.]
Remember that UNIX is a line-oriented operating system--really all the utilities work on text files, on a line by line basis.

Examples:

Supercomputer	scans for 'Supercomputer'
Super.*	any word containing 'Super'
[13579]	any of '1', '3', '5', '7', '9'
^Once upon a time	starting with 'Once upon a time'
[A-Z][a-z]*	any initial capitalized word
[qQ][u]	any 'q' or 'Q' followed by 'u'
^[^aeiou][aeiou]	matches word at start of line, begins with non-vowel and followed by a vowel.
\([0-9][0-9]*\)\2	matches second integer on a line

The above work in the UNIX command line, and in vi/vim

