Pointers to Functions (with thanks to David Southwell)

Use of this simple feature provides elegant solutions to some problems.

A pointer to a function holds the memory address of the function's executable code
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)

General form of Syntax is:

```
type  (*name)();
```

Consider the prototype for some function (here message):

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

we could declare the function pointer **fptr** as:

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

Initialize the pointer by simply assigning the name to it.

```
fptr = message;
```

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  (*respond_to_keypress[256])();

respond_to_keypress['\t']  =  tab_handler;

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

So now when some key c is pressed we invoke

```
respond_to_keypress[c](c);
```

and the job is done

Similarly we need to a pass function as a parameter:

• we want a callback from a graphical object.
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". 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

int qsort (char* base, int nel, int width, int (*compar)());

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

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

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

and a call to qsort might look like:

```
qsort ( a, 1024, sizeof float, compare_float );
```

See also UNIX Quicksort function.

Regular Expressions

Used for pattern-matching strings

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
. and single character
c Any character; use \c if special (like, ^, &, .)
[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

* After an expression, this will match n duplications (n can be zero, i.e. the NULL string)

x y expression x followed by expression y

\(...\) delimits a subexpression

\n where n in range 0-9, \n matches the n-th subexpression found in this line

UNIX grep command

grep <regex> <list of files>

this command scans all the characters in the files specified, and prints out every line that contains at least one match with the regular expression <regex> given.

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	Super.*
[13579]	^Once upon a time
[A-Z][a-z]*	[qQ][^u]
[^aeiou][aeiou]*	\([0-9][0-9]*\)\2

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

How about pattern matching from within a C program?

First one must set up the environment so that the dynamic linker can find the system libraries used for the manipulation of regular expressions

LD_LIBRARY_PATH /usr/ucblib

Modify the makefile so that the gcc knows about the libraries (object modules) that match patterns.

Use the flags **-L/usr/ucblib -lucb**

Since the necessary header files are not viewable from falun (for reasons that remain a mystery, you must also provide the necessary prototypes to keep the compiler happy

char* re_comp (char*);

int re_exex (char*);

To be useful pattern matching must be very fast, so the raw text form of the regular expressions require some parsing to convert them to a more efficient representation. It is **re_comp()** that does this. The **re_exex()** call is used to check the string against our regular expression.

Example program:

```
#include <stdio.h>

char* re_comp (char*);
int re_exex (char*);

void main (void) {
    char* return_value;
    char input_string [256]

    return_value = re_comp ("[A-Z]{a-z]*");

    if (return_value != (char*) NULL) {
        puts (return_value);
        exit (1);
    }
    do {
        scanf ("%s", input_string);
    } while (re_exex (input_string));

    printf ("%s Need a capitalized name!\n",
    input_string);
    exit (0);
}
```