Pointers to Functions

A simple idea, which leads to elegant solutions to several problems.

A pointer to a function holds the address of the function; when the pointer is dereferenced, the effect is a call to the function.

The pointer's type must match the return value of the function; however the input parameter types are not used.

Unlike data pointers, the only operations which can be performed on function pointers are :

- assignment to
- assignment from
- dereference (ie. call)

Pointers to Functions

C syntax :

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

So, given a function prototype :

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

...we could declare a pointer to this function like so :

```
int (*ptf)();
```

...and initialise it by simply assigning the name of the function to it:

```
ptf = initialise;
```

...and we can call this function through ptf like this :

```
retval = (*ptf)(64, "I love CMPUT 201");
```

Pointers to Functions

So what can we use them for ?

1. When we need to select a function to call at runtime, and switch blocks would be impractical or ugly.

For example, say we wanted to read a key press and call a different function depending on the value of the key...

Set up an array of pointers to functions...

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

respond_to_keypress['\t'] = tab;

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

And call them when a key is pressed...

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


Pointers to Functions

2. Passing functions as parameters

It is often convenient to use pointers to pass functions as parameters...

- callbacks from graphical objects

(please call this function when the user presses this button...)

- OS calls like sort and search, where you need to tell the system how to compare data types the system doesn't need to know about.
- functions in objects. As you will discover as you touch C++, classes are essentially structures which can include "member functions". It is straightforward to achieve most of this functionality in C using pointers to functions...

Pointers to Functions

Example : using UNIX Quicksort

All the sorting algorithms have at their heart an evaluation function, which is used to rank two elements.

The sort algorithm is independent of the details of this comparison, so why not write a generic sort routine, and call a function (passed as a parameter) each time we need to compare two items?

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

base	the start of the array
nel	number of elements in array
width	size of each element (bytes)
compar	pointer to function taking pointers to two elements, and returning -,0 or + (<,= or >)

Pointers to Functions

So, for sorting doubles, we might define a compare function thus :

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

...and a call to qsort might look like this :

```
qsort(a, 1024, 8, compare_double);
```

Regular Expressions

...are used for pattern matching strings.

We can use strcmp(char *, char *) for precise string comparison, but very often we would like to be able to find strings which conform to certain constraints.

Regular expressions prescribe a pattern in an array of characters. We can tell the system what our regular expression is, and then ask it if particular strings conform to it.

They are found throughout the UNIX environment, either at the command line (through GREP - Get REgular exPression), or through system calls from within user programs...as we shall see...

Regular Expressions

Some single character matching codes...

c	Any character; use \c if a special
^	denotes the start of the line
$	denotes the end of the line
.	any single character
[abc]	a, b or c are matched
[d-k]	d,e,f,g,h,i,j or k are matched
[^xyz]	any character not in the set...

Operators...

*	After an expression, this will match n duplications (n can be 0)
xy	expression x followed by expression y
\(...\)	delimits a subexpression
\n	n : 0-9 \n matches the nth sub-expression matched this line

Regular Expressions

UNIX grep

grep <regex> <files>

...scans all the characters in the files specified, and prints out each line where there is at least 1 match with the regular expression given.

eg.

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

Note that you will need to use ['] if your regular expression includes codes interpreted by the shell before grep sees them.

UNIX is a line oriented OS - nearly all the utilities work on text files, and on a line by line basis.

Regular Expressions

Some examples...

Supercomputer
Super.*
[13579]
^Once upon a time
[A-Z][a-z]*
[qQ][^u].*
\([0-9][0-9]*\)\1
[^aeiou][aeiuo]*

So, how do we access this from within our C programs ?

First, we must set up your environment so that the dynamic linker can find the system libraries for the manipulation of regular expressions...

LD_LIBRARY_PATH /usr/ucblib

Regular Expressions

Also, modify the makefile so that the appropriate libraries are visible from gcc.

Use the flags -L/usr/ucblib -lucb

It appears that the function headers are not available from falun. You should use the following to keep the type checker happy :

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

The pattern matching must be very fast, and the raw text form of the regular expressions require some parsing, to take them down to a more efficient representation. We use the re_comp() call to do this. The re_exec() call is used to check the string against our regular expression.

Regular Expressions

Example program :

```
#include <stdio.h>

char *re_comp(char *);
int re_exec(char *);

main()
{
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);
  }
```

Regular Expressions

```
do
  {
  scanf("%s", input_string);
  } while (re_exec(input_string));

printf("%s wasn't a capitalised name!\n",
        input_string);
exit(0);
}
```

The loop takes a string from the console, and checks it with the previously submitted regular expression; this will return a 1 for a match, or a 0 for no match. The loop continues until no match is found - the program terminates.

re_exec continues to use the initial regular expression until re_comp() is called with a new one.

Where to get these slides

Copies of these notes can be found from my web page :

http://web.cs.ualberta.ca/~daves

They are in compressed postscript form, so to print them out...

- download them from the web, by clicking on them.
- uncompress, with gunzip...

  gunzip 201_dave.ps.gz

- and print directly to a postscript printer...

  lpr -P<your printer> 201_dave.ps