cs3157 CGI lecture (mon-14-feb-2005)

- today:
  - continuing with C
    - control flow
    - strings
    - string library
  - CGI
    - cgi with C
    - cgi with perl
  - leftover perl
    - more built-in functions
    - subroutines
    - regular expressions
    - pattern matching

C control flow.

- *blocks* are enclosed in curly brackets
- functions are blocks
- `main()` is a function
- blocks have two parts:
  - variable declaration ("data segment")
  - code segment
- in C, variables have to be declared before they are used
- initializations can occur at the end of the declaration section, but before the code section

strings (1).

- storing multiple characters in a single variable
- data type is still `char`
- BUT it has a *length*
- last character the is *terminator*: `'\0'`, aka `NULL`
- string constants are surrounded by *double* quotes: `"`
- example:

```
char s[6] = "ABCDE";
```


strings (2).

- example:

```
char s[6] = "ABCDE";
```

- storage looks like this:

A	B	C	D	E	\0

- so with strings, you really only access the values stored at indeces 0 through $length - 2$, since the value stored at $length - 1$ is always `\0`

strings (3).

- printing strings
- format sequence: `%s`
- example:

```
#include <stdio.h>
int main() {
  char str[6] = "ABCDE";
  printf( "str = %s\n", str );
} /* end of main() */
```

- output:

```
ABCDE
```


string library (1).

- to use the string library, include the header in your C source file:

```
#include <string.h>
```

- string length function:

```
int strlen( char *s );
```

this function returns the number of characters in `s`; note that this is NOT the same thing as the number of characters allocated for the string array

- string comparison function:

```
int strcmp( const char *s1, const char *s2 );
```

"This function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared."

- for more information and more string functions, do (e.g.):

```
unix$ man strcmp
```


string library (2).

- copying functions:

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

  – copies characters from `source` array into `dest` array up to NULL

```
char *strncpy( char *dest, char *source, int num );
```

  – copies characters from `source` array into `dest` array; stops after `num` characters (if no NULL before that); appends NULL

string library (3).

- search functions:

```
char *strchr( const char *source, const char ch );
```

  – returns pointer to first occurrence of `ch` in `source`; NULL if none

```
char *strstr( const char *source, const char *search );
```

  – return pointer to first occurrence of `search` in `source`

string library (4).

- parsing function:

```
char *strtok( char *s1, const char *s2 );
```

  - breaks string `s1` into a series of *tokens*, delimited by `s2`
  - called the first time with `s1` equal to the string you want to break up
  - called subsequent times with NULL as the first argument
  - each time is called, it returns the next token on the string
  - returns null when no more tokens remain

```
char inputline[1024];
char *name, *rank, *serial_num;
printf( "enter name+rank+serial number: " );
scanf( "%s", inputline );
name = strtok( inputline,"+" );
rank = strtok( null,"+" );
serial_num = strtok( null,"+" );
```


string library (5).

- formatting functions — using internal buffers:

```
int sscanf(char *string, char *format, ...)
```

  - parse the contents of string according to format
  - placed the parsed items into 3rd, 4th, 5th, ... argument
  - return the number of successful conversions

```
int sprintf(char *buffer, char *format, ...)
```

  - produce a string formatted according to format
  - place this string into the buffer
  - the 3rd, 4th, 5th, ... arguments are formatted
  - return number of successful conversions

- format characters are like `printf` and `scanf` (see notes from earlier lectures)

CGI (1).

- CGI = common gateway interface
- standard for invoking external applications from within a browser
- basis for things like PHP
- name the executable SOMETHING.cgi (use the “cgi” extension)
- output — first line MUST be:
  - `Content-type: text/plain` for plain text output
  - `Content-type: text/html` for html output
- input — two ways (via html forms):
  - environment (`GET`)
  - stdin (`POST`)
- with C, compile like this (e.g.):

```
gcc  -o plain_c.cgi  plain_c.c
```


CGI (2).

- examples: `http://www1.cs.columbia.edu/~cs3157/cgi`
- plain text output:
  `plain_c.cgi` (from `plain_c.c`) and
  `plain_pl.cgi`
- html output:
  `html_c.cgi` (from `html_c.c`) and
  `html_pl.cgi`
- env input:
  `form_qs_c.html` to `qs_c.cgi` (from `qs_c.c`) and
  `form_qs_pl.html` to `qs_pl.cgi`
- stdin input:
  `form_stdin_c.html` to `stdin_c.cgi` (from `stdin_c.c`) and
  `form_stdin_pl.html` to `stdin_pl.cgi`

perl —subroutines.

- syntax for defining:

```
sub name {block}
sub name (proto) {block}
```

- where `proto` is like a prototype, where you put in sample arguments
- syntax for calling:

```
name(args);
name args;
```

- any arguments passed to a subroutine come in as the array `@_`
- you can use the `return` statement, like in C

perl —built-in functions.

- here are a few:
- `chomp $var`
  `chomp @list`
  removes any line-ending characters
- `chop $var`
  `chop @list`
  removes last character
- `chr number`
  returns the character represented by the ASCII value number
- `eof filehandle`
  returns true if next read on filehandle will return end-of-file
- `exists $hash{$key}`
  returns true if specified hash key exists, even if its value is undefined
- `exit`
  exits the perl process immediately

- `getc filehandle`
  reads next byte from filehandle
- `index string, substr [, start]`
  returns position of first occurrence of substr in string, with optional starting position; also `rindex` which is index in reverse
- `opendir dirhandle, dirname`
  opens a directory for processing, kind of like a file; use `readdir` and `closedir` to process
- `split /pattern/, string [, limit]`
  splits string into a list of substrings, by finding delimiters that match pattern;
  example: `split /([-,])/,"1-10,20";` returns `(1, '-', 10, ',', 20)`
- `substr string, pos [, n, replacement]`
  returns substring in string starting with position pos, for n characters

perl —regular expressions.

- simplest regular expression is a literal string
- complex regular expressions use *metacharacters* to describe various options in building a pattern... *"I never metacharacter I didn't like"*
- metacharacters:

`\`	escapes the character immediately following it
`.`	matches any single character except newline
`^`	matches at the beginning of a string
`$`	matches at the end of a string
`*`	matches the preceding element 0 or more times
`+`	matches the preceding element 1 or more times
`?`	matches the preceding element 0 or 1 times
`{ ... }`	specifies a range of occurrences for the element preceding it
`[ ... ]`	matches any one of the class of characters in the brackets
`( ... )`	groups expressions
`|`	matches either the expression before or after it

note that there are some exceptions to these rules

perl —pattern matching.

- `=~` binds a scalar to a patterm match, substitution or translation
- `!~` just like above, except that the return value is negated in the logical sense
- operators:
  - `m/pattern/gimosx` : match
    - g = match globally (all instances)
    - i = do case insensitive matching
    - note that first m is optional
  - `s/pattern/replacement/egimosx` : search
    - e = evaluate right side as an expression
    - g = match globally (all instances)
    - i = do case insensitive matching
  - `y/pattern1/pattern2/cds` : translate
    - c = complement pattern1
    - d = delete found but unreplaced characters
    - s = squash duplicate replaced characters

perl —pattern matching, example 1.

- example

```
#!/usr/bin/perl

$s = "hello world";
print '$s=[',$s,"]\n";

if ($s =~ m/x/) { print "there's an x in ",$s,"\n" }
else { print "there isn't\n" }

if ($s =~ m/L/i) { print "there's an l in ",$s,"\n" }
else { print "there isn't\n" }
```

- output:

```
$s=[hello world]
there isn't
there's an l in hello world
```


perl —pattern matching, example 2.

- example

```
#!/usr/bin/perl

$s = "hello world";
print '$s=[',$s,"]\n";

$t = ($s =~ s/l/x/g);
print '$t=[',$t,"]\n";
print '$s=[',$s,"]\n";
```

- output:

```
$s=[hello world]
$t=[3]
$s=[hexxo worxd]
```


perl —pattern matching, example 3.

- example

```
#!/usr/bin/perl

$s = "hello world";
print '$s=[',$s,"]\n";

$u = ($s =~ y/l/o/c);
print '$u=[',$u,"]\n";
print '$s=[',$s,"]\n";
```

- output:

```
$s=[hello world]
$u=[8]
$s=[oolooooolo]
```