ECS 36A, May 23, 2024

Last C Operator

- Abbreviated “if”

```
x = a ? b : c
```

- If `a` evaluates to non-zero, `b` is evaluated and assigned to `x`
  - `c` is ignored
- If `a` evaluates to zero, `c` is evaluated and assigned to `x`
  - `b` is ignored

Examples

```
a = 0;
b = 1;
c = 2;
x = a ? b++ : c--;
```

As `a` = 0, `c--` is evaluated, so

`x` = 2 and `c` = 1

```
a = 3;
b = 1;
c = 2;
x = a ? b++ : c--;
```

As `a` ≠ 0, `b++` is evaluated, so

`x` = 1 and `b` = 2

Function Pointers

- Pointers are addresses
- Functions are in memory, and so have addresses
- So a function pointer contains the address of a function
- Example declaration:

```
int (*func)(char *)
```

this points to a function that takes a character pointer as an argument and returns an integer

Example Usage

```
int add(int x)  { return(x + 4); }
int sub(int y)  { return(y – 4); }

…
int main(void)
{
   int (*f)(int);
   …
   f = add;
   z = f(5);
   …
   f = sub;
   z = f(5);
   …
```

Background

- System calls: interfaces to operating system functions
- Example: some Linux system calls
  - I/O: reading, writing, networking, etc.
  - Files: chown, chgrp, stat, etc.
  - Resource usage: ulimit, getrlimit, etc.
  - Timing: gettimeofday, time
- Library functions provide system-independent interface to them
  - Also provide other features

C Library Functions

- The C library provides many functions that do useful things
  - Standard I/O C library
  - Math library
- Character type
- String to integer or float/double types
- Handling options
- Time
- Random numbers
- String and memory manipulation

Standard I/O Functions

- Implements open, read, write, close, and others
- *Requires* #include <stdio.h>
- Basis: streams or files
  - Usually FILE * types
  - Buffers input, output
  - Predefined streams: stdin (input), stdout (output), stderr (error output)

Buffering

- For efficiency; goal is to reduce number of read, write system calls
- On read, the library reads a block of data
    - The number of bytes in a block here depends on the system
    - This is *not* the same thing as a block in a program; it’s a chunk of data
- The library then returns the amount of data requested, and keeps the rest in memory
- On next library call, it returns the next byte *without* doing another call to system
- This explains why *ungetc*() can only guarantee one char of pushback

Full Buffering in Standard I/O Library

- Typically used when reading/writing files
- Read: call to system call fills buffer; next call is when a read occurs and buffer is empty
- Write: call to system call empties buffer; next call is when a write occurs and the buffer is full
- Flushing: emptying the buffer; as noted, done automatically
    - Use *fflush*() to do this manually
- On exit or return from *main*(), all buffers are flushed

Line Buffering in Standard I/O Library

- Typically used with line-oriented devices such as terminals
- Buffers flushed when newline encountered *or* buffer is full
    - Doesn’t matter if buffer is for reading or for writing
    - Also output is flushed when process reads from a line-buffered or unbuffered stream
- Idea is to act like fully buffered I/O, except that reading/writing in blocks is infeasible, as process can’t read a terminal beyond what has been typed
- On exit or return from *main*(), all buffers are flushed

Unbuffered Streams in Standard I/O Library

- Don’t buffer anything
- On input, byte *immediately* made available to process
  - Terminals usually need to be put into a special mode (called ``raw’’ mode) in which no character processing is done; usual mode is called ``sane’’ or ``cooked’’
- On output, character is *immediately* written to device or file

Useful Functions: Positioning for Read/Write

- Every stream has a *read/write pointer* (*rw-pointer*) pointing to where the next byte is to be read or written
- fgetpos(*fp*, *pos*): gets current position *pos* of rw-pointer of *fp*
    - ftell(*fp*, *pos*): return position of rw-pointer of *fp*
- fsetpos(*fp*, *pos*): set current position *pos* of rw-pointer of *fp*
    - rewind(*fp*): reset rw-pointer to 0 (the beginning of the file)
- fseek(*fp*, *offset*, *whence*): set current position of rw-pointer of *fp* to *offset* bytes from *whence*
    - *whence* is SEEK_SET (beginning), SEEK_CUR (current position), or SEEK_END (from the end)
- ftell(*fp*): return location of rw-pointer of *fp*

More C Library Functions

- time
- (pseudo)random numbers
- string functions
- memory functions
- math functions

Get Time

- Use system call time_t time(time_t **tick*)
  - If *tick* is NULL, then the current time is returned
  - Time measured in seconds from the epoch (Jan 1, 1970, 00:00:00)
- To get time as a string: char *ctime(&*tick*)
  - On success, generates a string of the following form:

    Sun Sep 16 01:03:52 1973

    (This has a trailing nnewline)
  - On failure, it returns NULL

Time Structure

```
struct tm {
   int tm_sec;       /* 0-59 seconds */
   int tm_min;       /* 0-59 minutes */
   int tm_hour;      /* 0-23 hour */
   int tm_mday;      /* 1-31 day of month */
   int tm_mon;       /* 0-11 month */
   int tm_year;      /* 0- year - 1900 */
   int tm_wday;      /* 0-6 day of week (Sunday = 0) */
   int tm_yday;      /* 0-365 day of year */
   int tm_isdst;     /* flag: daylight savings time in effect */
            /* the following are not present on all systems */
   long tm_gmtoff;   /* offset from GMT in seconds */
   char **tm_zone;   /* abbreviation of time zone */
};
```

Getting Structure Values for Time

- struct tm *localtime(const time_t **timep*): fills in local time
- struct tm *gmtime(const time_t **timep*): fills in GMT (UTC) time
  - Here *timep* is a pointer to what *time* returns
- char *asctime(struct tm **tm*): return a ctime-type string for *tm*
- time_t mktime(struct tm *tm): return time since the epoch given by *tm*

Random Numbers

- int rand(void)
  - Generate pseudorandom number between 0 and RAND_MAX inclusive
  - **This function is dangerous — avoid it!!** In older versions, it is *not* pseudorandom in the low order bits. (On newer Linux systems, it's OK)
- long random(void)
  - Generate pseudorandom number between 0 and 2^{31}–1 inclusive
- All require a starting point – called a *seed*

Random Number Seeds

- void srand(unsigned int seed)
  - Initialize the *rand*() pseudorandom number generator with *seed*
- void srandom(unsigned int seed)
  - Initialize the *random*() pseudorandom number generator with *seed*
- Pick *seed* as randomly as possible
- There are defaults, useful for regenerating the same sequence for debugging
  - rand/srand default seed is 1
  - random/srandom default seed is 1