Embedded Software

2. C programming

Koen Langendoen
Embedded Software Group
C crash course

- For Java programmers
  - Main differences
  - Common pitfalls

- Language + tools // next lecture

- Learning by doing
  - Online – Weblab
  - Hands on – supervised labs
C for Java Programmers*

Henning Schulzrinne
Dept. of Computer Science
Columbia University

*Selection and editing by Koen Langendoen
C history

- C
  - Dennis Ritchie in late 1960s and early 1970s
  - *systems* programming language
    - make OS portable across hardware platforms
    - not necessarily for real applications – could be written in Fortran or PL/I

- C++
  - Bjarne Stroustrup (Bell Labs), 1980s
  - object-oriented features

- Java
  - James Gosling in 1990s, originally for embedded systems
  - object-oriented, like C++
  - ideas and some syntax from C
Why learn C (after Java)?

- Both high-level and low-level language
  - OS: user interface to kernel to device driver
- Better control of low-level mechanisms
  - memory allocation, specific memory locations
- Performance *sometimes* better than Java
  - usually more predictable
- Most older code is written in C
- Being multi-lingual is good!

- But,....
  - Memory management responsibility
  - Explicit initialization and error detection
  - generally, more lines for same functionality

Ideal for embedded systems

More room for errors
Prog. language popularity

[Graph showing the popularity of different programming languages from 2002 to 2014. The graph is titled 'TIOBE Programming Community Index' and indicates changes in popularity over time for languages like Java, C, C++, Objective-C, C#, Python, JavaScript, Visual Basic, and PHP.]
## C vs. Java

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>object-oriented</td>
<td>function-oriented</td>
</tr>
<tr>
<td>strongly-typed</td>
<td>can be overridden</td>
</tr>
<tr>
<td>polymorphism (+, ==)</td>
<td>very limited (integer/float)</td>
</tr>
<tr>
<td>classes for name space</td>
<td>(mostly) single name space</td>
</tr>
<tr>
<td>macros are external, rarely used</td>
<td>macros common (preprocessor)</td>
</tr>
<tr>
<td>layered I/O model</td>
<td>byte-stream I/O</td>
</tr>
</tbody>
</table>
# C vs. Java

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic memory management</td>
<td>By hand: function calls <em>(malloc, free)</em></td>
</tr>
<tr>
<td>no pointers (only references)</td>
<td>points (memory addresses) common</td>
</tr>
<tr>
<td>by-reference, by-value</td>
<td>by-value parameters</td>
</tr>
<tr>
<td>exceptions, exception handling</td>
<td>if (f() &lt; 0) {error} OS signals</td>
</tr>
<tr>
<td>concurrency (threads)</td>
<td>library functions</td>
</tr>
</tbody>
</table>
# C vs. Java

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>length of array</td>
<td>on your own</td>
</tr>
<tr>
<td>string as type</td>
<td>just bytes (char []), with 0 end</td>
</tr>
<tr>
<td>dozens of common libraries</td>
<td>OS-defined</td>
</tr>
</tbody>
</table>
Java program

- collection of classes
- class containing main method is starting class
- running `java StartClass` invokes `StartClass.main` method
- JVM loads other classes as required
C program

- collection of functions
- one function – `main()` – is starting function
- running executable (default name a.out) starts main function
- typically, single program with all user code linked in – but can be dynamic libraries (.dll, .so)
C vs. Java

public class hello
{
    public static void main (String args []) {
        System.out.println(“Hello world”);
    }
}

#include <stdio.h>

int main(int argc, char *argv[])
{
    puts(“Hello world\n”);
    return 0;
}
Executing a C program

```c
int main(int argc, char *argv[])
```

- `argc` is the argument count
- `argv` is the argument vector
  - array of strings with command-line arguments
- the `int` value is the return value
  - convention: 0 means success, > 0 some error
  - can also be declared as void (no return value)
Simple example

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello World.\n");
    for(int i=0; i<argc; i++)
        printf("\t arg %d: %s\n", i, argv[i]);
    return 0;
}

$ a.out
Hello World.
    arg 0: a.out
$
```
Executing a C program

- Name of executable + space-separated arguments
- `$ a.out 1 23 'third arg'`
The C compiler gcc

- gcc invokes C compiler
- gcc translates C program into executable for some target
- default file name a.out
- also “cross-compilation”

$ gcc hello.c
$ a.out
Hello, World!
Behavior controlled by command-line switches:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o file</td>
<td>output file for object or executable</td>
</tr>
<tr>
<td>-Wall</td>
<td>all warnings – use always!</td>
</tr>
<tr>
<td>-c</td>
<td>compile single module (non-main)</td>
</tr>
<tr>
<td>-g</td>
<td>insert debugging code (gdb)</td>
</tr>
<tr>
<td>-p</td>
<td>insert profiling code</td>
</tr>
<tr>
<td>-l</td>
<td>library</td>
</tr>
<tr>
<td>-E</td>
<td>preprocessor output only</td>
</tr>
<tr>
<td>-std=c99</td>
<td>C++ style comments, local vars in for loops, ...</td>
</tr>
</tbody>
</table>
Using gcc

- Two-stage compilation
  - pre-process & compile: `gcc -c hello.c`
  - link: `gcc -o hello hello.o`

- Linking several modules:
  ```
  gcc -c a.c \rightarrow a.o
  gcc -c b.c \rightarrow b.o
  gcc -o hello a.o b.o
  ```

- Using math library
  ```
  gcc -o calc calc.c -lm
  ```
Error reporting in gcc

- If `gcc` gets confused, hundreds of messages
  - fix first, and then retry – ignore the rest
- `gcc` will produce an executable with warnings
  - don’t ignore warnings – compiler choice is often not what you had in mind
- Does not flag common mindos
  - `if (x = 0)` vs. `if (x == 0)`
The C preprocessor (cpp) is a macro-processor which
- manages a collection of macro definitions
- reads a C program and transforms it

Example:

```c
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)

if (check(i)) {...}
```

becomes

```c
const int MAXVALUE = 100;
int check(int x) {
    return x < MAXVALUE;
}

if (((i) < 100)) {...}
```
C preprocessor

- Preprocessor directives start with # at beginning of line:
  - define new macros (don’t try this at home! 😋)
  - input files with C code (typically, definitions)
  - conditionally compile parts of file
- `gcc -E` shows output of preprocessor
- Can be used independently of compiler
C preprocessor - file inclusion

#include “filename.h”
#include <filename.h>

- inserts contents of filename into file to be compiled
- “filename” relative to current directory
- <filename> relative to /usr/include
- gcc -I flag to re-define default
- import function prototypes (cf. Java import)

Examples:

#include <stdio.h>
#include “mydefs.h”
#include “/home/alice/program/defs.h”
C preprocessor – conditional compilation

#if expression
code segment 1
#else
code segment 2
#endif

- preprocessor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- machine or OS-dependent code
- can be used to comment out chunks of code – bad!

#define OS linux
...
#if OS == linux
    puts("Linux!");
#else
    puts("Something else");
#endif
C language

- Data model
  - simple, low-level

- Control structures
  - syntax quite similar to Java
  - sequencing: ;
  - grouping: {...}
  - selection: if, switch
  - iteration: for, while
  - operators: =, ==, +=, ++, &&, &

consistent indentation please!
### Numeric data types

<table>
<thead>
<tr>
<th>type</th>
<th>precision</th>
<th>#include &lt;stdint.h&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits</td>
<td>int8_t</td>
</tr>
<tr>
<td>short</td>
<td>≥ 16 bits</td>
<td>int16_t</td>
</tr>
<tr>
<td>int</td>
<td>≥ 16 bits</td>
<td>int32_t</td>
</tr>
<tr>
<td>long</td>
<td>≥ 32 bits</td>
<td>int64_t</td>
</tr>
<tr>
<td>long long</td>
<td>≥ 64 bits</td>
<td>int128_t</td>
</tr>
<tr>
<td>float</td>
<td>≥ 32 bits</td>
<td>IEEE 754 single prec.</td>
</tr>
<tr>
<td>double</td>
<td>≥ 64 bits</td>
<td>IEEE 754 double prec.</td>
</tr>
</tbody>
</table>

**Architecture dependent**

- preferred
Unsigned integers

- Also, unsigned versions of integer types
  - e.g., unsigned short, uint16_t

- same bits, different interpretation
  - shift right (>>) with(out) sign extension
    - ((int8_t)0xFF) >> 4 == 0xFF
    - ((uint8_t)0xFF) >> 4 == 0x0F
  - overflow is undefined for signed ints, but wrap-around for unsigned ints
    - ((uint8_t)0xFF) + 1 == 0x00

thou shalt avoid unsigneds
Type conversion

```c
#include <stdio.h>
void main(void)
{
    int i, j = 12;     /* i not initialized, only j */
    float f1, f2 = 1.2;

    i = (int) f2;     /* explicit: i <- 1, 0.2 lost */
    f1 = i;           /* implicit: f1 <- 1.0 */

    f1 = f2 + (float) j; /* explicit: f1 <- 1.2 + 12.0 */
    f1 = f2 + j;      /* implicit: f1 <- 1.2 + 12.0 */
}
```
Explicit and implicit conversions

- Implicit: e.g., \( s = i + c \)
- Promotion: char -> short -> int -> ...
- If one operand is double, the other is made double
- If either is float, the other is made float, etc.
- Explicit: type casting – \((\text{type})\)
- Almost any conversion does something – but not necessarily what you intended
Type conversion

int x = 100000;
short s;

s = x;
printf("%d %d\n", x, s);

100000  -31072
C - no booleans

- C doesn’t have booleans
- Emulate as int or char, with values 0 (false) and non-zero (true)
- Allowed by flow control statements:
  ```c
  if (n = 0) {
      printf("something wrong");
  }
  ```
- Assignment returns zero -> false
User-defined types

- typedef gives names to types:

  typedef short int smallNumber;
  typedef unsigned char byte;
  typedef char String[100];

  smallNumber x;
  byte b;
  String name;
Defining your own boolean

typedef char boolean;
#define FALSE 0
#define TRUE 1

- Generally works, but beware:
  check = x > 0;
  if (check == TRUE) {...}

- If \( x \) is positive, check will be non-zero, but may not be 1.
Enumerated types

- Define new integer-like types as enumerated types:
  ```c
  typedef enum {
      Red, Orange, Yellow, Green, Blue, Violet
  } Color;
  enum weather {rain, snow=2, sun=4};
  ```

- Look like C identifiers (names)
- Are listed (enumerated) in definition
- Treated like integers
  - Can add, subtract – even `color + weather`
  - Can’t print as symbol (unlike Pascal)
  - But debugger generally will
Enumerated types

- Just syntactic sugar for ordered collection of integer constants:

  ```
  typedef enum {
    Red, Orange, Yellow
  } Color;
  ```

  is like

  ```
  #define Red 0
  #define Orange 1
  #define Yellow 2
  ```

- `typedef enum {False, True} boolean;`
Objects (or lack thereof)

- C does not have objects / classes
  - but does support abstract data types through separate files
  - declaration (xxx.h) vs. implementation (xxx.c)
- Variables for C’s primitive types are defined similarly:
  ```c
  short int x;
  char ch;
  float pi = 3.1415;
  float f, g;
  ```
- Variables defined in `{}` block are active only in block
- Variables defined outside a block are global (persist during program execution), but may not be globally visible (static)
Data objects

- Variable = container that can hold a value
  - in C, pretty much a CPU word or similar
- default value is (mostly) undefined – treat as random
  - compiler may warn you about uninitialized variables
- \texttt{ch = ‘a’; x = x + 4;}
- Always pass by value, but can pass address to function:
  
  \texttt{scanf(“\%d\%f”, &x, &f);}
Data objects

- Every data object in C has
  - a name and data type (specified in definition)
  - an address (its relative location in memory)
  - a size (number of bytes of memory it occupies)
  - visibility (which parts of program can refer to it)
  - lifetime (period during which it exists)

- Warning:

```c
int *foo(char x) {
    return &x;
}

pt = foo(x);
*pt = 17;
```
Data objects

- Unlike scripting languages and Java, all C data objects have a fixed size over their lifetime
  - except dynamically created objects
- size of object is determined when object is created:
  - global data objects at compile time (data)
  - local data objects at run-time (stack)
  - dynamic data objects by programmer (heap)
Data object creation

```c
int x;
int arr[20];
void main(int argc, char *argv[]) {
    int i = 20;
    {int x; x = i + 7;}
}
void f(int n)
{
    int a, *p;
    a = 1;
    p = (int *)malloc(sizeof int);
}
Data object creation

- `malloc()` allocates a block of memory
- Lifetime until memory is freed, with `free()`
- Memory *leakage* – memory allocated is never freed:

```c
char *combine(char *s, char *t) {
    u = (char *)malloc(strlen(s) + strlen(t) + 1);
    if (s != t) {
        strcpy(u, s); strcpy(u+strlen(s), t);
        return u;
    } else {
        return NULL;
    }
}
```
Memory allocation

- **Note:** `malloc()` does not initialize data
- `void *calloc(size_t nmemb, size_t size)` does initialize (to zero)
  - `malloc(sz) \approx calloc(sz, 1)`
Memory layout of programs

- Header info
- Code
- Data - Heap
- Data - stack
- Dynamic memory
- Local memory + function call stack

All `malloc()`s
All normal vars
Data objects and pointers

- The memory **address** of a data object, e.g., `int x`
  - can be obtained via `&x`
  - has a data type `int *` (in general, `type *`)
  - has a value which is a large (4/8 byte) unsigned integer
  - can have pointers to pointers: `int **`

- The **size** of a data object, e.g., `int x`
  - can be obtained via `sizeof x` or `sizeof(x)`
  - has data type `size_t`, but is often assigned to `int` (bad!)
  - has a value which is a small(ish) integer
  - is measured in bytes
Data objects and pointers

- Every data type T in C has an associated pointer type T *
- A value of type T * is the address of an object of type T
- If an object int *xp has value &x, the expression *xp dereferences the pointer and refers to x, thus has type int

```
xp
&x
int *

int
42
```
Data objects and pointers

- If p contains the address of a data object, then *p allows you to use that object
- *p is treated just like normal data object

```c
int a, b, *c, *d;
*d = 17; /* BAD idea */
a = 2; b = 3; c = &a; d = &b;
if (*c == *d) puts("Same value");
*c = 3;
if (*c == *d) puts("Now same value");
c = d;
if (c == d) puts("Now same address");
```
void pointers

- Generic pointer
  
  void *malloc(size_t size);
  void free(void *ptr);

- Unlike other pointers, can be assigned to any other pointer type:
  
  void *v = malloc(13);
  char *s = v;

- Acts like char * otherwise:
  
  v++, sizeof(*v) = 1;
Structured data objects

- Structured data objects are available as

<table>
<thead>
<tr>
<th>object</th>
<th>property</th>
</tr>
</thead>
<tbody>
<tr>
<td>array []</td>
<td>enumerated, numbered from 0</td>
</tr>
<tr>
<td>struct</td>
<td>names and types of fields</td>
</tr>
<tr>
<td>union</td>
<td>occupy same space (one of)</td>
</tr>
</tbody>
</table>
Arrays

- Arrays are defined by specifying an element type and number of elements
  - `int vec[100];`
  - `char str[30];`
  - `float m[10][10];`

- Stored as linear arrangement of elements

- For array containing $N$ elements, indexes are $0..N-1$
  - `int sum = 0;
    for (int i = 0; i < N; i++)
      sum += vec[i];`
Arrays

- C does not remember how large arrays are (i.e., no length attribute)
  - no out-of-bounds checking
  - int x[10]; x[10] = 5; may work (for a while)
- In the block where array A is defined:
  - sizeof A gives the number of bytes in array
  - can compute length via sizeof A / sizeof A[0]
- When an array is passed as a parameter to a function
  - the size information is not available inside the function
  - array size is typically passed as an additional parameter
    - PrintArray(A, VECSIZE);
  - or globally
    - #define VECSIZE 10
Copying arrays

- Copying content vs. copying pointer to content

```c
void copy(int A[], int B[], int N) {
    A = B;
}
```

- Swizzling pointers has no effect, copy contents element-wise instead

```c
void copy(int A[], int B[], int N) {
    for (int i = 0; i < N; i++) {
        A[i] = B[i];
    }
}
```
Strings

- In Java, strings are regular objects
- In C, strings are just `char` arrays with a **null** (`\0`) terminator

```
   a c a t \0
```

- “a cat” =
- A literal string (“a cat”)
  - is automatically allocated memory space to contain it and the terminating `\0`
  - has a value which is the address of the first character
  - can’t be changed by the program (common bug!)
- All other strings must have space allocated to them by the program
Strings

- We normally refer to a string via a pointer to its first character:
  ```c
  char str[] = "my string";
  char *s;
  s = &str[0]; s = str;
  ```

- C functions only know string ending by \0:
  ```c
  char *str = "my string";
  for (int i = 0; str[i] != '\0'; i++)
    putchar(str[i]);
  for (char *s = str; *s != '\0'; s++)
    putchar(*s);
  ```

- String library: `#include <strings.h>`
  - `strlen`, `strcpy`, ...
structs

- Similar to fields in Java object/class definitions
- Components can be any type (but not recursive)
- Accessed using the same syntax `struct.field`
- Example:

  ```
  struct {int x; char y; float z;} rec;
  ...
  rec.x = 3; rec.y = 'a'; rec.z = 3.1415;
  ```
structs

- Record types can be defined
  - using a tag associated with the struct definition
  - wrapping the struct definition inside a typedef

- Examples:
  ```c
  struct complex { double real; double imag; };
  struct point { double x; double y; } corner;
  typedef struct { double real; double imag; } Complex;
  struct complex a, b;
  Complex c, d;
  ```

- `a` and `b` have the same size, structure and type
- `a` and `c` have the same size and structure, but different types
Dereferencing pointers to 
struct elements

- Pointers commonly to structs
  
  ```c
  Complex *p;
  double i;

  (*p).real = 42.0;
  i = (*p).imag;
  ```

- Note: `*p.real` doesn’t work

- Abbreviated alternative:
  
  ```c
  p->real = 42.0;
  i = p->imag;
  ```
Functions

- Prototypes and functions (cf. Java interfaces)
  - extern int putchar(int c);
  - putchar('A');
  - int putchar(int c) {
      do something interesting here
  }
- If defined before use in same file, no need for prototype
- Typically, prototype defined in .h file
- Good idea to include <.h> in actual definition
Functions

- static functions and variables hide them to those outside the same file:
  ```java
  static int x;
  static int times2(int c) {
    return c*2;
  }
  ```

- compare protected class members in Java.
Program with multiple files

- Library headers
  - Standard
  - User-defined

#include <stdio.h>
#include "mypgm.h"

void main(void)
{
  myproc();
}

main.c

#include "mypgm.h"

void myproc(void);

mypgm h

#include "mypgm.h"

static int mydata;

void myproc(void)
{
  mydata=2;
  ... /* some code */
}

mypgm c
Data hiding in C

- C doesn’t have classes or private members, but this can be approximated

- Header file defines public data:
  ```c
  typedef struct queue_t *queue_t;
  queue_t NewQueue(void);
  ```

- Implementation defines real data structure:
  ```c
  #include “queue.h”
  // good practice
  typedef struct queue_t {
      struct queue_t *next;
      int data;
  } *queue_t;

  queue_t NewQueue(void) {
      return calloc(1, sizeof(struct queue_t)); // with 0 contents
  }
  ```
Function pointers

- functions can be used as values (i.e. passed by reference)

```c
int foo();    // function returning integer
int *bar();   // function returning pointer to int
int (*fp)();  // pointer to function returning int
int *(*fpp)();// pointer to func returning ptr to int

fp = foo;
fpp = bar;

int i = fp();
int j = *(fpp());
```
Function pointers

- to install interrupt handlers (timers, etc)

```c
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

- to register call back functions
- to implement polymorphism
Before we break ....

- Always initialize anything before using it (especially pointers)
- Don’t use pointers after freeing them
- Don’t return a function’s local variables by reference
- No exceptions – so check for errors everywhere
  - memory allocation
  - system calls
  - Murphy’s law, C version: anything that can’t fail, will fail
- An array is also a pointer, but its value is immutable.
Programming State Machines

- Finite State Machines
  - prime design pattern in embedded systems
- Transitions initiated by events
  - interrupts (timers, user input, ...)
  - polling
- Actions
  - output
  - modifying system state (e.g., writing to global variables)
Running example

- See Wikipedia: Automata-based programming\(^1\)

- Consider a program in C that reads a text from the standard input stream, line by line, and prints the first word of each line. Words are delimited by spaces.

\(^1\)[https://en.wikipedia.org/wiki/Automata-based_programming]
Exercise (5 min)

• Consider a program in C that reads a text from the standard input stream, line by line, and prints the first word of each line. Words are delimited by spaces.

Code
```c
#include <stdio.h>
#include <ctype.h>

int main(void) {
    int c;
    do {
        do
            c = getchar();
        while (c == ' ');
        while (c != ' ' && c != '
' && c != EOF) {
            putchar(c);
            c = getchar();
        }
        putchar('
');
        while (c != '
' && c != EOF)
            c = getchar();
    } while (c != EOF);
    return 0;
}
```
S = space
N = newline
A = all other chars
* = print

FSM

before

inside

after

S

A*

N*

A, S

done

EOF
int main(void) {
    enum states {
        before, inside, after
    } state;
    int c;
    state = before;
    while((c = getchar()) != EOF) {
        switch(state) {
            case before:
                if(c != ' ') {
                    putchar(c);
                    if(c != '
')
                        state = inside;
                }
                break;
            case inside:
                break;
            case inside:
                // FSM-based solution
                // 1 loop
                // 1 case for EOF checking
        }
    }
}
```c
17. case inside:
18.     if(c == ' ')
19.         state = after;
20.     else if(c == '\n') {
21.         putchar('\n');
22.         state = before;
23.     } else
24.         putchar(c);
25.     break;
26. case after:
27.     if(c == '\n') {
28.         putchar('\n');
29.         state = before;
30.     }
31.     break;
32. default:
33.     fprintf(stderr, "unknown state %d\n", state);
34.     abort();
```

**defensive programming!**
enum states { before, inside, after }
void step(enum states *state, int c)
{
    switch(*state) {
        case before: ... *state = inside; ...
        case inside: ... *state = after; ...
        case after: ... *state = before; ...
    }
}

int main(void)
{
    int c;
    enum states state = before;
    while((c = getchar()) != EOF) {
        step(&state, c);
    }
    return 0;
}
Function pointers

1. enum states { before, inside, after };  
2. void step(enum states *state, int c)  
3. {  
4.     switch(*state) {  
5.         case before:  
6.             if(c != ' ')  
7.                 putchar(c);  
8.                 if(c != '\n')  
9.                     *state = inside;  
10.             }  
11.             break;  
12.         case inside:  
13.             if(c == ' ')  
14.                 *state = after;  
15.             else if(c == '\n')  
16.                 putchar('\n');  
17.                 *state = before;  
18.             }  
19.         } else
20. }
void (*state)(int c); // global function-ptr variable :-(

void before(int c) { ... }
void inside(int c) { ... }
void after(int c) {
    if(c == '\n') {
        putchar('\n');
        state = before;
    }
}

int main(void)
{
    int c;
    state = before;
    while((c = getchar()) != EOF) {
        (*state)(c);
    }
    return 0;
}

• function per state
Function pointers

1. `statefp before(int c) { ... }`
2. `statefp inside(int c) { ... }
3. `statefp after(int c) {
4.     if(c == '\'n') {
5.         putchar('\n');
6.         return before;
7.     }
8.     else
9.         return after;
10. }
11.int main(void)
12.{
13.    int c;
14.    statefp state = before;
15.    while((c = getchar()) != EOF) {
16.        state = (*state)(c);
17.    }
18.    return 0;
Function pointers

1. typedef void (*voidfp)(); // hack around recursive definition
2. typedef voidfp (*statefp)(int c);
3. voidfp after(int c) {
4.    if(c == '\n') {
5.        putchar('\n');
6.        return (voidfp) before;
7.    }
8.    else
9.        return (voidfp) after;
10.}
11.int main(void)
12.{
13.    int c;
14.    statefp state = before;
15.    while((c = getchar()) != EOF) {
16.        state = (statefp) (*state)(c);
17.    }
18.    return 0;

• return values vs global variable

Solution

No recursive typedefs, so void * to the rescue¹

¹http://www.gotw.ca/gotw/057.htm
FSM: table-based solution

- Transition:
  - action
  - next state

1. int main(void)
2. {
3.   int c;
4.   states state = before;
5.   while((c = getchar()) != EOF) {
6.     edges edge = lookup(state, c);
7.     edge.action();
8.     state = edge.next;
9.   }
10.  return 0;
11.}
Lookup tables

- Case dispatch
  - if-then-else
  - switch
  - table

```c
1. voidfp inside(int c) {
2.     if(c == ' ')
3.         return after;
4.     else if(c == '
') {
5.         putchar(c);
6.         return before;
7.     } else {
8.         putchar(c);
9.         return inside;
10. }
```

```c
5. voidfp lookup[] = {
6.     /* space */ after,
7.     /* newline */ before,
8.     /* other */ inside};
```

```c
5. voidfp inside(int c) {
6.     return lookup[c];
7. }
```
FSM: table-based solution

- Transition:
  - action
  - next state

1. int main(void)
2. {
3.   int c;
4.   states state = before;
5.   while((c = getchar()) != EOF) {
6.     edges *edge = &lookup[state][c];
7.     edge->action(c);
8.     state = edge->next;
9.   }
10. return 0;
11.}
Function per Transition

1. void skip(int c) {
2. }
3. void print(int c) {
4.     putchar(c);
5. }

6. typedef void (*actions)(int c);
7. typedef enum {before, inside, after, num_states} states;
8. typedef enum {space, newline, other, num_inputs} inputs;
9. typedef struct {states next; actions act;} edges;

10. edges lookup[num_states][num_inputs] = {
11.     /* space        newline         others */
12.     /* before */ {{before,skip}, {before,print}, {inside,print}},
13.     /* inside */ {{after, skip}, {before,print}, {inside,print}},
14.     /* after  */ {{after, skip}, {before,print}, {after, skip} }
15.};
Function per Transition

1.   edges lookup[num_states][num_inputs] = {
2.     /* space       newline       others */
3.     /* before */ {{before,skip}, {before,print}, {inside,print}},
4.     /* inside */ {{after, skip}, {before,print}, {inside,print}},
5.     /* after  */ {{after, skip}, {before,print}, {after, skip} }
6.   };
7.   int main(void)
8.   { int c;
9.     states state = before;
10.    while((c = getchar()) != EOF) {
11.       inputs inp = char2inp(c);
12.       edges *edge = &lookup[state][inp];
13.       edge->act(c);
14.       state = edge->next;
15.    }
16.   return 0;
17.}
18.}

inputs char2inp(char c) {
    if (c == ' ')
        return space;
    else if (c == '\n')
        return newline;
    else
        return other;
}