Embedded Software

CSE2425

2. C programming

Koen Langendoen
Embedded and Networked Systems

C crash course

- For Java programmers
  - Main differences
  - Common pitfalls
- Language + tools // next^3 lecture
- Learning by doing
  - Online – Weblab
  - TA support – Queue

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)?

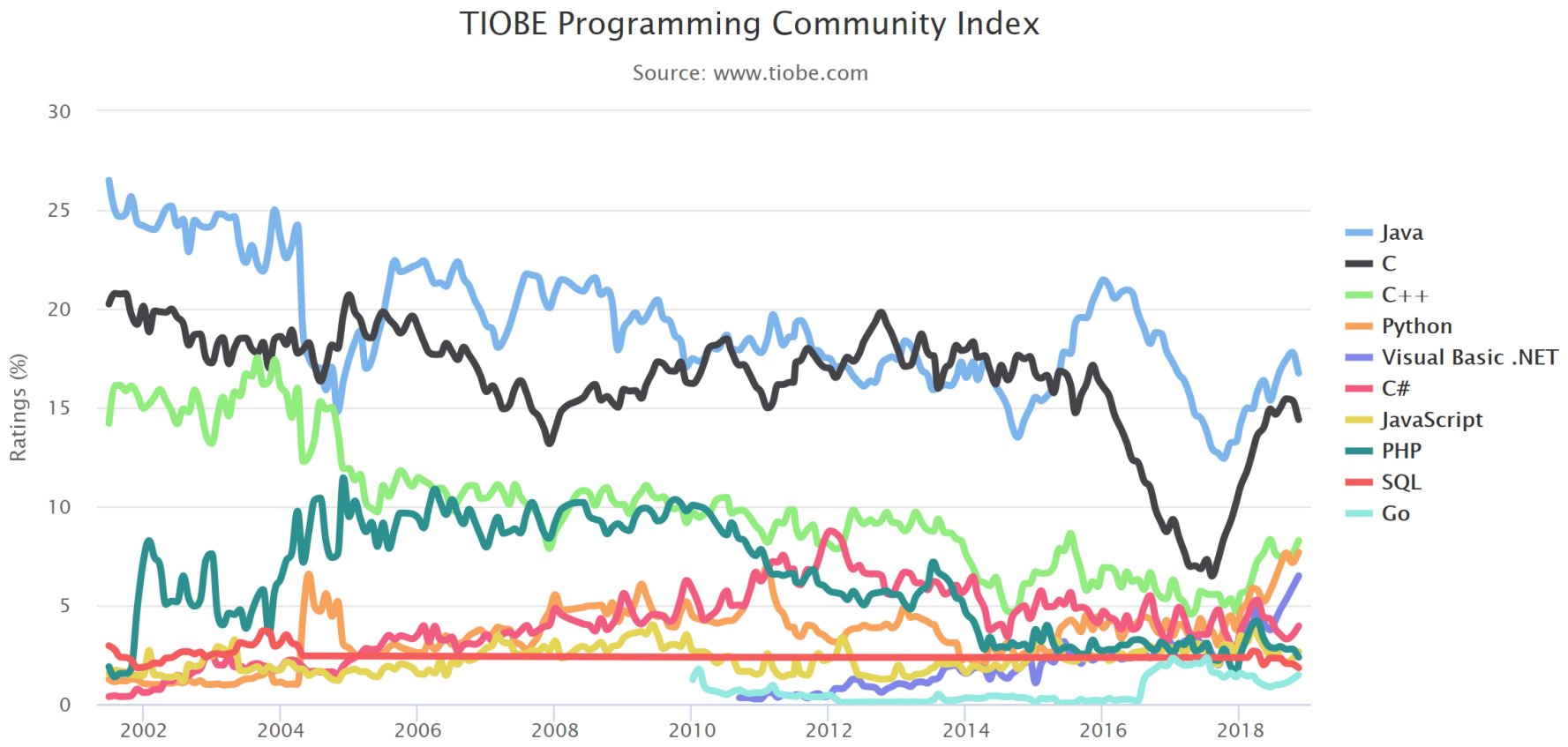
- Both high-level and low-level language
  - OS: user interface to kernel to device driver
- Better control of low-level mechanisms
  - memory allocation, specific memory locations
- Performance *sometimes* better than Java
  - usually more predictable
- Most older code is written in C
- Being multi-lingual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

Ideal for embedded systems

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

More room for errors

Prog. language popularity



C vs. Java

Java	C
object-oriented	**function-oriented**
strongly-typed	can be overridden
polymorphism (+, ==)	very limited (integer/float)
classes for name space	(mostly) single name space
macros are external, rarely used	macros common (preprocessor)
layered I/O model	byte-stream I/O

C vs. Java

Java	**C**
automatic memory management	By hand: function calls (**malloc, free**)
no pointers (only references)	**pointers** (memory addresses) common
by-reference, by-value	by-value parameters
exceptions, exception handling	if (f() < 0) {error} OS signals
concurrency (threads)	library functions

C vs. Java

Java	**C**
length of array	**on your own**
string as type	just bytes (char []), with 0 end
dozens of common libraries	OS-defined

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;
}
```

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 C programs

- Scripting languages are usually interpreted
  - perl (python, Tcl) reads script, and executes it
  - sometimes, just-in-time compilation – invisible to user
- Java programs semi-interpreted:
  - javac converts `foo.java` into `foo.class`
  - not machine-specific
  - *byte codes* are then interpreted by JVM
- C programs are normally compiled and linked:
  - gcc converts `foo.c` into `a.out`
  - `a.out` is executed by OS and hardware

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!
```

gcc

- Behavior controlled by command-line switches:

-o *file*	output file for object or executable
-Wall	all warnings – use always!
-c	compile single module (non-main)
-g	insert debugging code (gdb)
-p	insert profiling code
-l	library
-E	preprocessor output only
-std=c99	C++ style comments, local vars in for loops, …

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` → a.o

  `gcc -c b.c` → 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)`

C preprocessor

- The C preprocessor (cpp) is a macro-processor that
  - manages a collection of macro definitions
  - reads a C program and transforms it
  - Example:

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

if (check(i)) {…}
```

becomes

```
if (((i) < 100))  {…}
```

```
const int MAXVALUE = 100;

int check(int x) {
    return x < MAXVALUE;
}
```

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
```

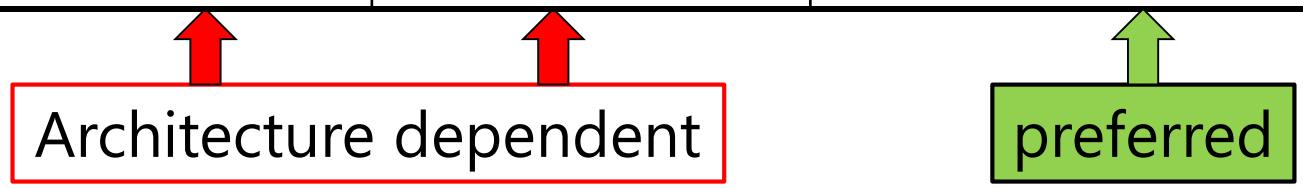
C language

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

consistent **indent**ation please!

Numeric data types

type	precision	#include <stdint.h>
char	8 bits	int8_t
short	≥ 16 bits	int16_t
int	≥ 16 bits	int32_t
long	≥ 32 bits	int64_t
long long	≥ 64bits	int128_t
float	≥ 32 bits	IEEE 754 single prec.
double	≥ 64 bits	IEEE 754 double prec.



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

```
#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 – (*`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:

```
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:

```
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:

```
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
- `ch = 'a'; x = x + 4;`
- Always pass by value, but can pass address to function:

```
scanf("%d%f", &x, &f);
```

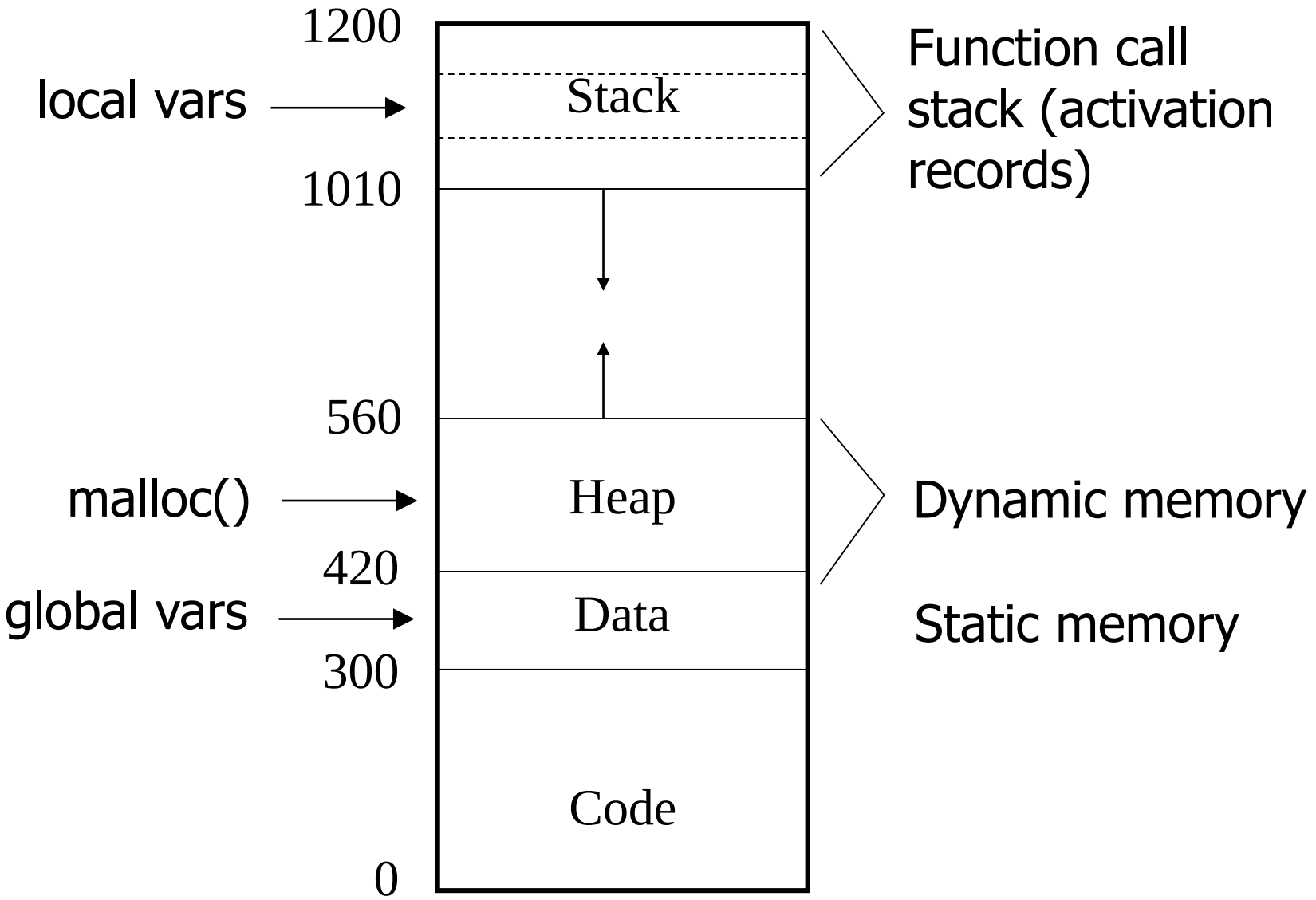
Data objects

- 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)

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)

Memory layout of programs



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:

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

dangling pointer (ouch!)

Data objects

- Warning:

```
int *foo(char x) {
  return &x;
}
void main() {
  char *pt;
  pt = foo('a');        putc(*pt);
  *pt = 'b';            putc(*pt);
  foo('c');             putc(*pt);
}
```

dangling pointer (ouch!)

- Output
  - A – abb
  - B – abc
  - C – <segmentation fault>

Data object creation

```
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:

```
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

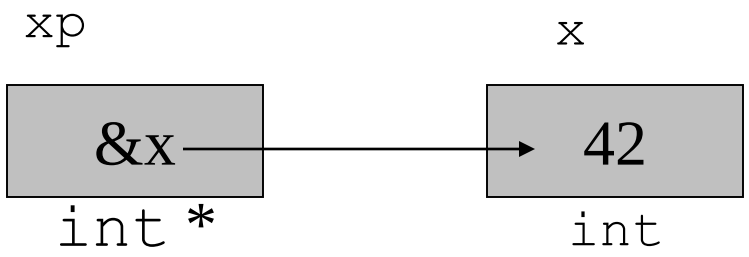
- Note: `malloc()` does not initialize data
- `void *calloc(size_t nmemb, size_t size)` does initialize (to zero)
  - `malloc(sz) ≈ calloc(sz, 1)`

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`



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

```
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:

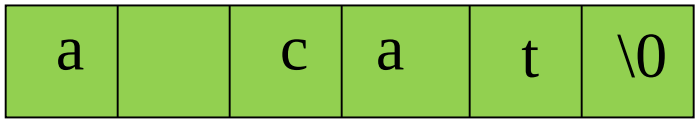
```
v++, sizeof(*v) = 1;
```

Structured data objects

- Structured data objects are available as

object	property
`array []`	enumerated, numbered from 0
`struct`	names and types of fields
`union`	occupy same space (one of)

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

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

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

```
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 NUL (‘\0’) terminator
- “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:

```
char str[] = "my string";
char *s;
s = &str[0]; s = str;
```

- C functions only know string ending by \0:

```
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:

```
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`

```
Complex *p;
double i;

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

- Note: `*p.real` doesn't work
- Abbreviated alternative:

```
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:

```
static int x;
static int times2(int c) {
  return c*2;
}
```

- compare protected class members in Java.

Program with multiple files

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

void main(void)
{
  myproc();
}
```

main.c

```
void myproc(void);
```

mypgm.h

```
#include "mypgm.h"

static int mydata;

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

mypgm.c

- Library headers
  - Standard
  - User-defined

Data hiding in C

- C doesn't have classes or private members, but this can be approximated
- Header file defines public data:

```
typedef struct queue_t *queue_t;
queue_t NewQueue(void);
```

- Implementation defines real data structure:

```
#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)

```
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)

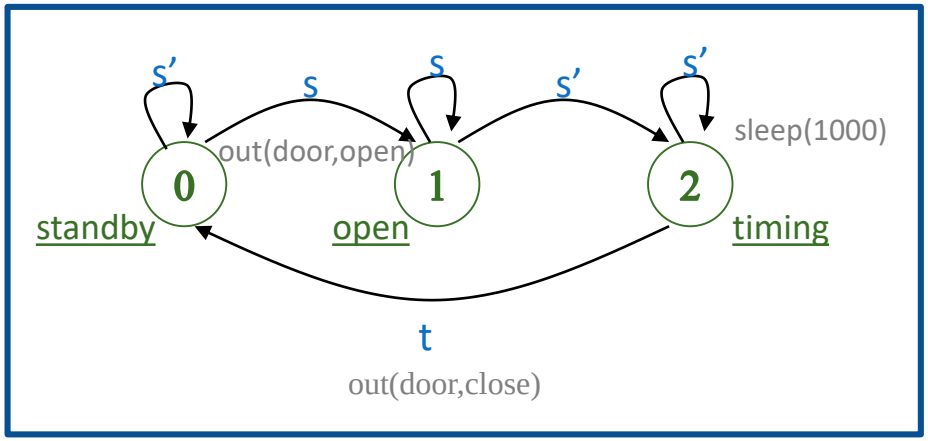
```
#include <signal.h>

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)

Code

- 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.

Ad-hoc solution

- too many loops
- duplicate EOF corner casing

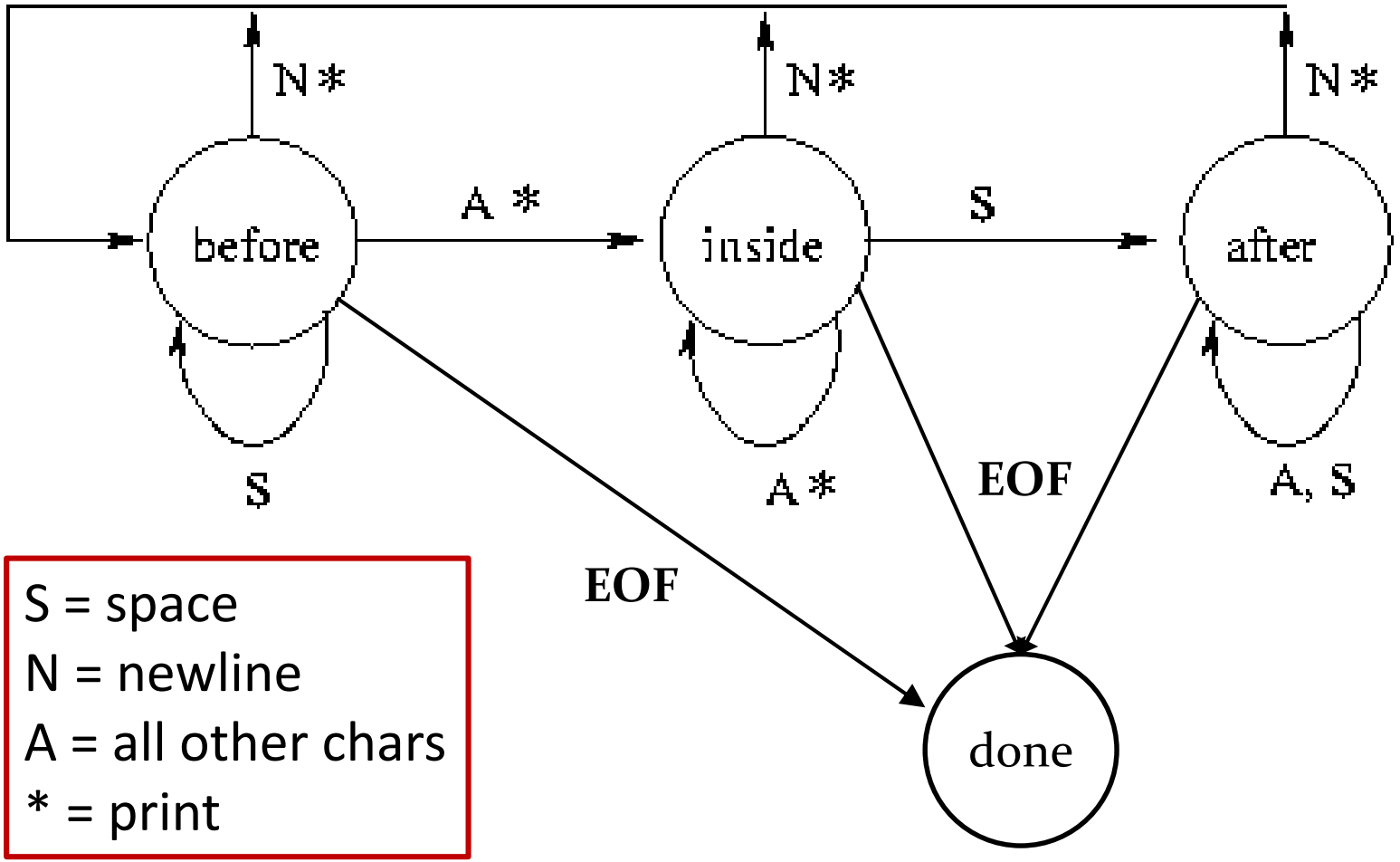
```
1.  #include <stdio.h>
2.  #include <ctype.h>
3.  int main(void)
4.  {
5.      int c;
6.      do {
7.          do
8.              c = getchar();
9.          while(c == ' ');
10.         while(c != ' ' && c != '\n' && c != EOF) {
11.             putchar(c);
12.             c = getchar();
13.         }
14.         putchar('\n');
15.         while(c != '\n' && c != EOF)
16.             c = getchar();
17.     } while(c != EOF);
18.     return 0;
19. }
```

skip leading spaces

print word

skip trailing chars

FSM



S = space
N = newline
A = all other chars
* = print

FSM-based solution

```
1.  int main(void)
2.  {
3.      enum states {
4.          before, inside, after
5.      } state;
6.      int c;
7.      state = before;
8.      while((c = getchar()) != EOF) {
9.          switch(state) {
10.             case before:
11.                 if(c != ' ') {
12.                     putchar(c);
13.                     if(c != '\n')
14.                         state = inside;
15.                 }
16.                 break;
17.             case inside:
```

- 1 loop
- 1 case for EOF checking

FSM-based solution

```
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!

Refactored solution

```
1. enum states { before, inside, after };
2. enum states step(enum states state, int c)
3. {
4.     switch(state) {
5.         case before: ... state = inside; ...
6.         case inside: ... state = after; ...
7.         case after:  ... state = before; ...
8.     }
9.     return state;
10.}
11.int main(void)
12.{
13.    int c;
14.    enum states state = before;
15.    while((c = getchar()) != EOF) {
16.        state = step(state, c);
17.    }
18.    return 0;
19.}
```

- lifted loop

Function pointers

```
1. enum states { before, inside, after };
2. enum states 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.            } else
```

- wanted: 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;
19.}
```

Exercise

Provide a typedef for statefp

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;
19.}
```

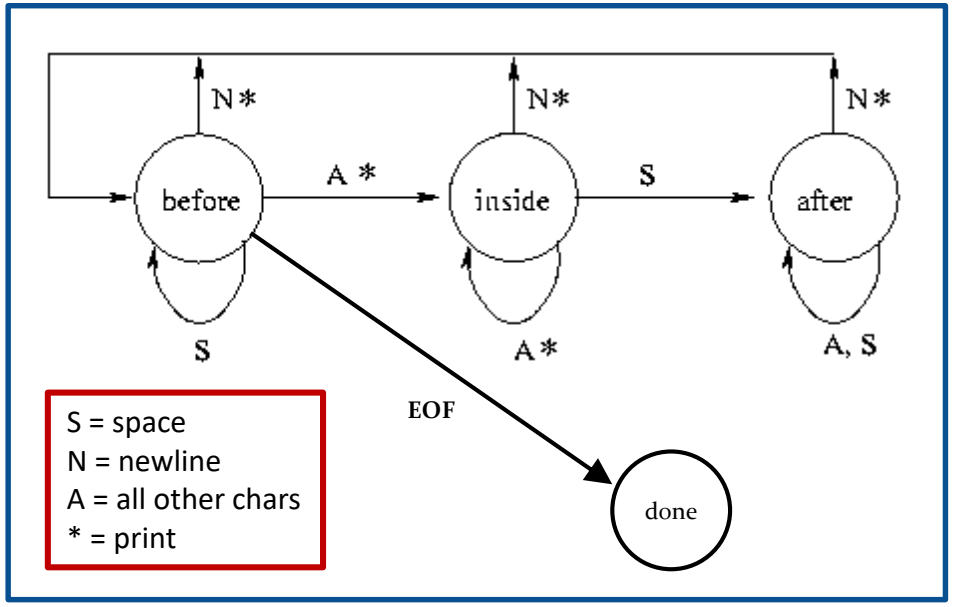
Solution

No recursive typedefs, so void * to the rescue[1]

[1] http://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(c);
8.         state = edge.next;
9.      }
10.     return 0;
11. }
```

Lookup tables

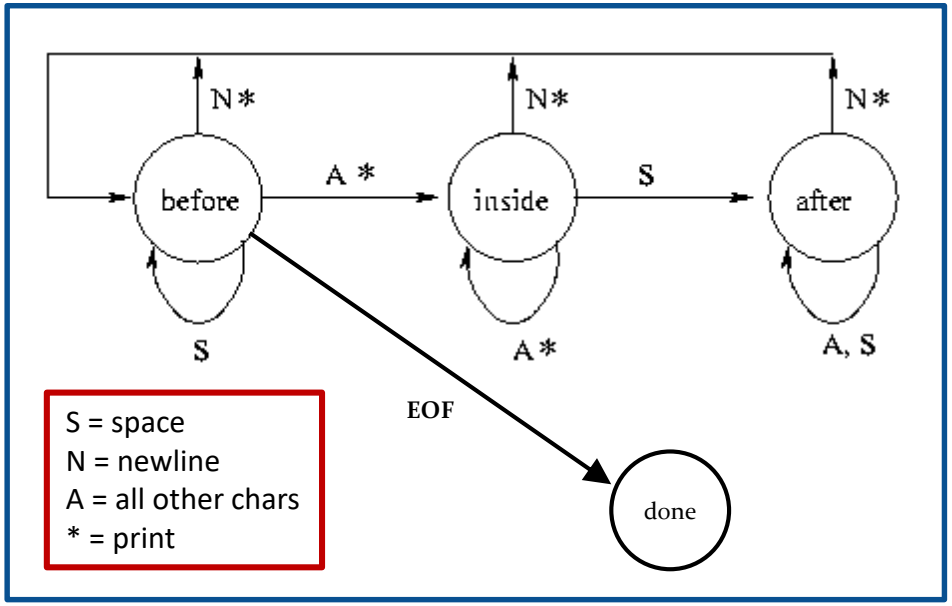
- Case dispatch
  - if-then-else
  - switch
  - table

```
1. states lookup[] = {
2.     /* space */   after,
3.     /* newline */ before,
4.     /* other */   inside};

5. states inside(int c) {
6.     return lookup[c];
7. }
```

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

FSM: table-based solution

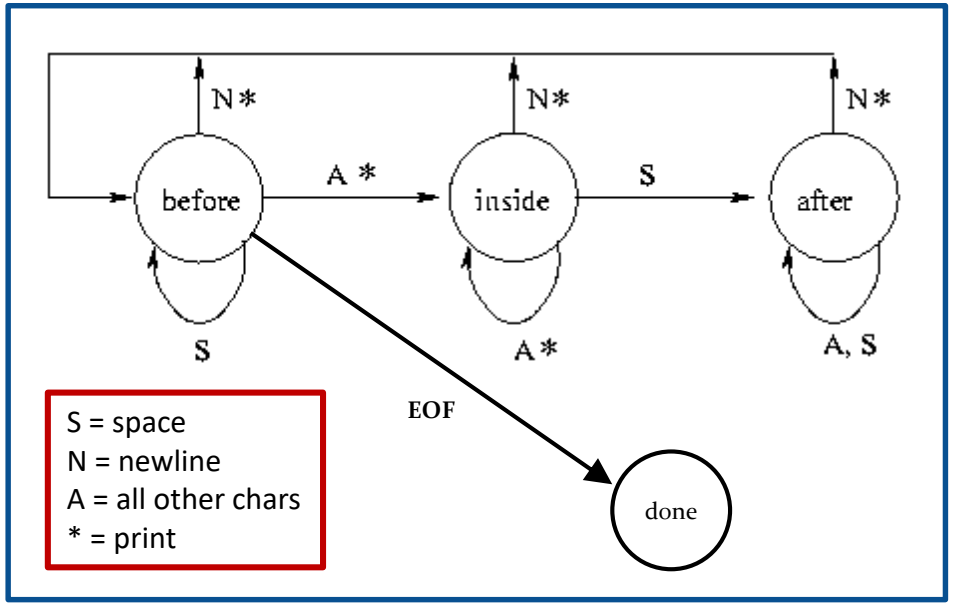
- Transition:
  - action
  - next state




```
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. }
```

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          other */
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          other */
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.  {
9.      int c;
10.     states state = before;
11.     while((c = getchar()) != EOF) {
12.         inputs inp = char2inp(c);
13.         edges *edge = &lookup[state][inp];
14.         edge->act(c);
15.         state = edge->next;
16.     }
17.     return 0;
18. }
```

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