Operating Systems
C Programming Refresh

Jyrki Katajainen ⟨`jyrki@di.ku.dk`⟩

Promised by Jost:

pointers, `malloc`, `free`, essential data structures

Actual contents:

motivation * C machine * pointers * types * kernel C * Turing-computable functions * syntactic sugar * memory management * pointers in action: stack * macros * inline assembly * programming style * idioms * reflection

Motivation

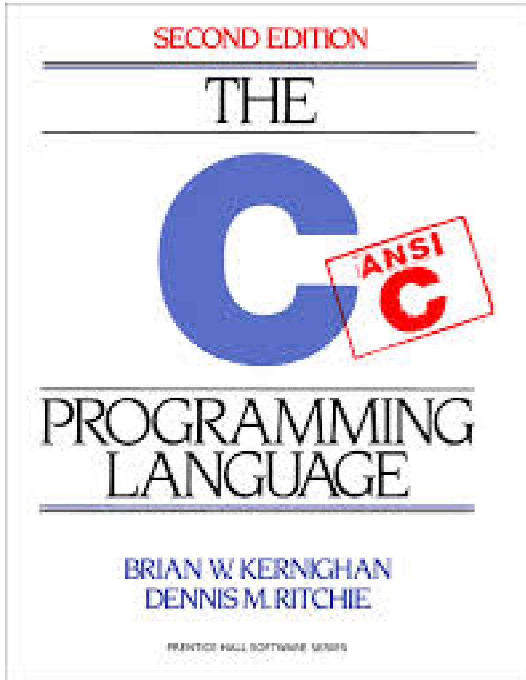
You should know C so well that you can understand the operating-system concepts explained in the book!

C code

```
list_del(&p->list);
kfree(p);
```

Reading

- Brian W. Kernighan & Dennis M. Ritchie, *The C Programming Language* (1988)



Optional

- Brian W. Kernighan & Rob Pike, *The Practice of Programming* (1999)
- Dennis M. Ritchie, The development of the C language, *ACM SIGPLAN Notices* (1993)
- S. Sandeep, *GCC-Inline-Assembly-HOWTO* (2003)

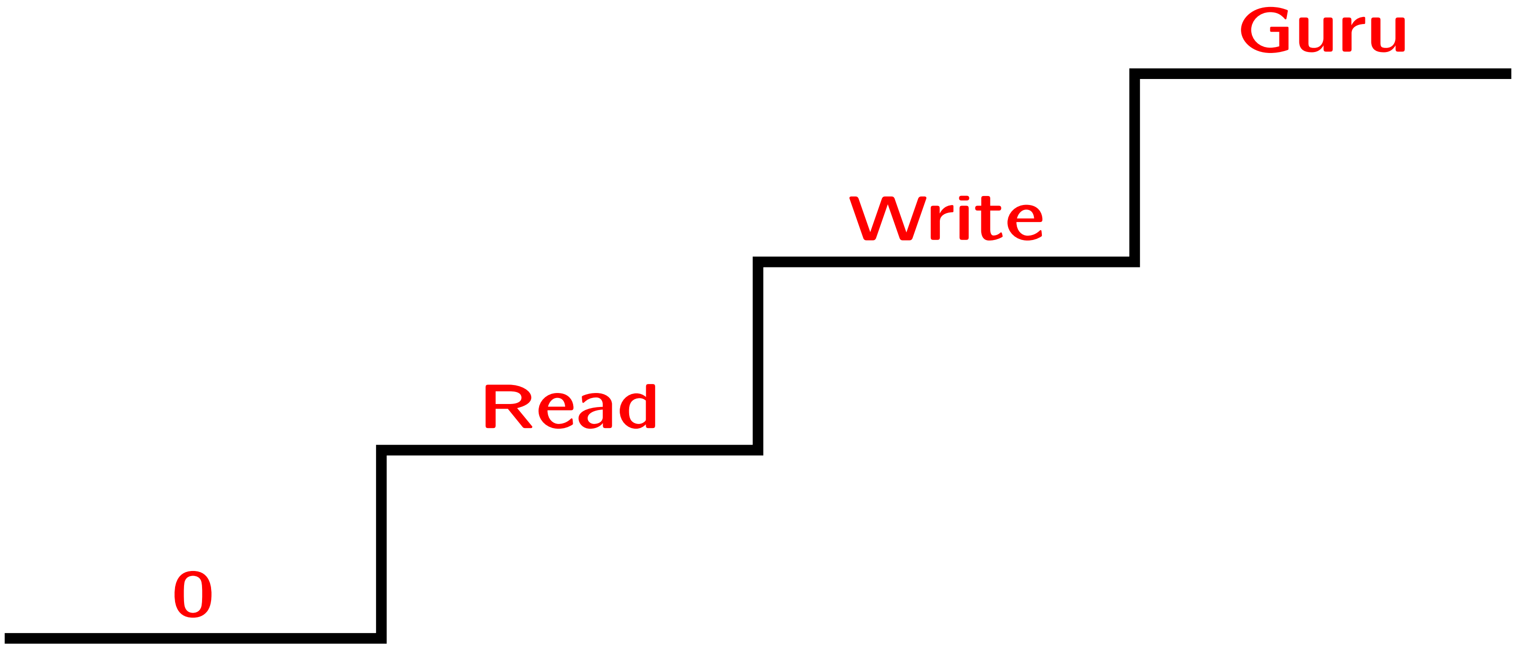
Disclaimer

I will concentrate on the fundamental issues, not any specific details. You have to read the textbook and practise to learn the language.

E.g. I will skip the following topics:

- bit manipulation [K&R, Section 2.9]
- command-line arguments [K&R, Section 5.10]
- structures [K&R, Ch. 6]
- input and output [K&R, Ch. 7]
- Unix interface [K&R, Ch. 8]
- library resources [K&R, App. B]

Knowledge levels



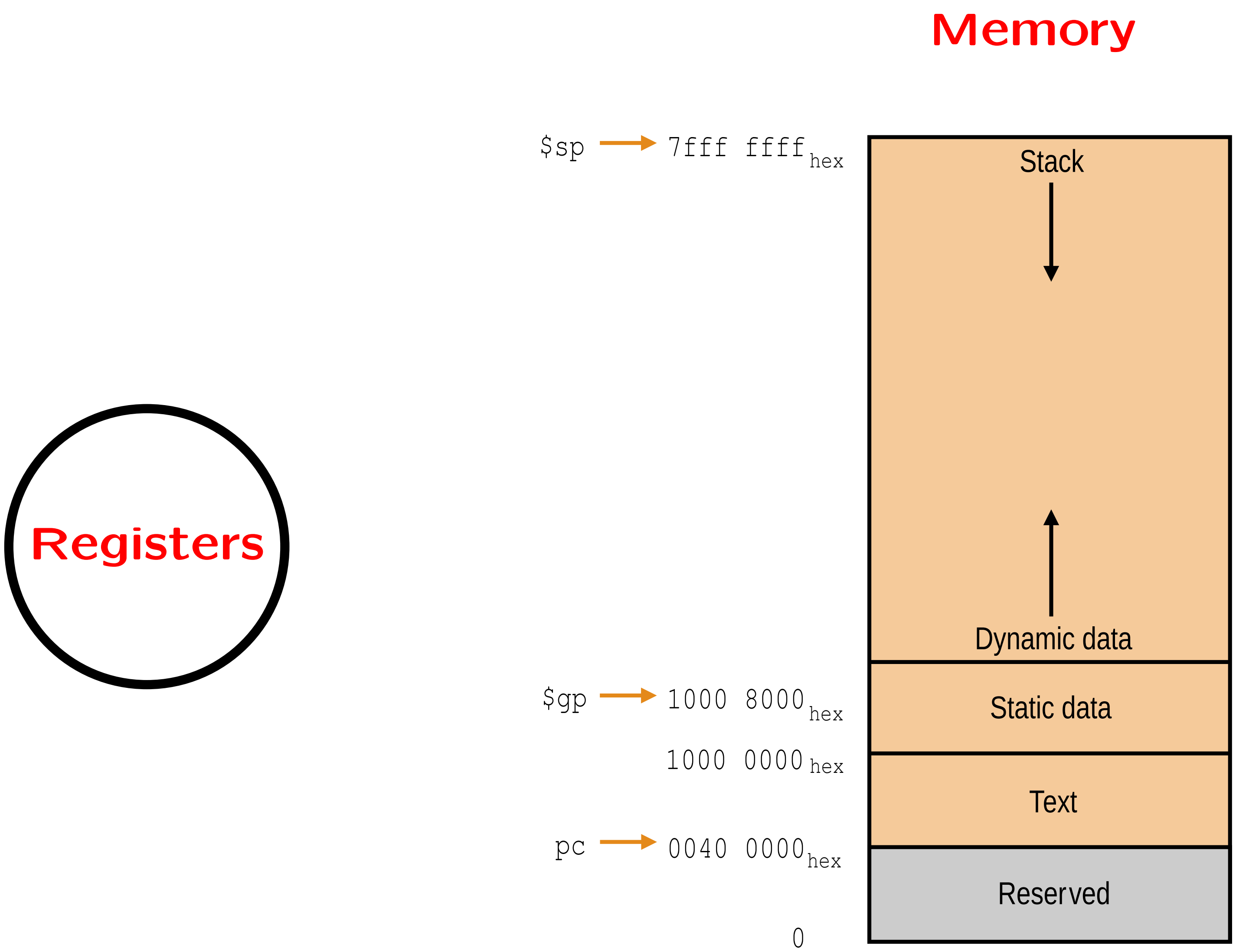
Skills you need

- Read old code
- Read badly-written code
- Read code written in a strange way
- Read low-level assembly code

Guru: Replace “Read” with “Debug and correct”

```
jyrki@Janus$ cat 100.c
_(__,___,____){___/__<=1?_(__,___+1,____):!(___%__)?_(__,___+1,0):___%__
==___/
__&&!____?(printf("%d\t",___/__),_(__,___+1,0)):___%__>1&&___%__<___/__?
_(__,1+
___,____+!(___/__%(___%__))):___<__*__?_(__,___+1,____):0;}main(){_(100,
0,0);}
jyrki@Janus$ gcc 100.c
100.c: In function ‘_’:
100.c:3:12: warning: incompatible implicit declaration of function ‘printf’
 __&&!____?(printf("%d\t",___/__),_(__,___+1,0)):___%__>1&&___%__<___/__?
            ^
jyrki@Janus$ ./a.out
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97
```

C machine



Memory
Registers
$sp
7fff ffff hex
Stack
Dynamic data
$gp
1000 8000 hex
Static data
1000 0000 hex
Text
pc
0040 0000 hex
Reserved
0

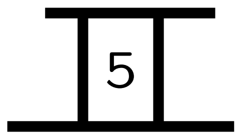
Pointers

A **pointer** is a variable that contains the address of a variable.

Address: Universitetsparken 5
Building at that address: HCØ

`p`**:** 0x7fff30788da4
`*p`**:** 5

0x7fff30788da4




```
int x = 5;
int* p = &x;
assert(*p == 5);
```

Types

In C, data declarations are read from right to left (and from the inside out). Keep this in mind when writing data declarations; `typedefs` can be used to improve readability.

`int const* p;` Pointer to a constant integer, i.e. the integer pointed to does not change.

`int* const q;` Constant pointer to an integer, i.e. the pointer is kept constant but the underlying integer can be modified.

```
typedef char* routine(char*, char const*);
routine* copy = copy_string;
```

Online exercise

What can you say about the following function signature?

```
#include <stdint.h>
...
TID_t thread_create(void (*func)(uint32_t), uint32_t arg)
```

[Buenos roadmap 2012, p. 19]

Warning: Your compiler understands C99 or C11, whereas our textbook describes ANSI C completed in 1989.

Expressions

- Expressions combine variables and constants to produce new values.
- Every expression has a value.
- The value of a relational or logical expression is 1 (true) or 0 (false)
- `0` means false and any other value means true.
- Be aware of the evaluation order of operators in an expression!

- `x = 0` is an expression
- `x = 0;` is a statement with the terminator `;`
- `x = 0` has the value `0` (the value of the left-hand side after the assignment)

Kernel of the C language

A *kernel-C program* is a sequence of statements that are executed sequentially unless the order is altered by a `while` statement.

The kernel of the C language has

- assignments,
- `while` statements, and
- nothing else.

Observe that a sequence {`statement`$_1$ `statement`$_2$ ...} is also a statement.

`x`: variable; `y`, `z`: variable | constant

`p`: pointer variable

$\mathcal{A}$: {`+`, `-`, `*`, `/`, `%`} (arithmetic)

$\mathcal{B}$: {`&`, `|`, `^`, `<<`, `>>`} (bitwise)

$\mathcal{C}$: {`<`, `<=`, `==`, `!=`, `>`, `>=`} (comparison)

$\mathcal{U}$ = {`-`, `~`, `&`} (unary)

Load: `x = *p;`

Store: `*p = y;`

Move: `x = y;`

Unary operation: $\ominus \in \mathcal{U}$

`x =` $\ominus$`y;`

Binary operation: $\oplus \in \mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$

`x = y` $\oplus$ `z;`

While loop: $\lhd \in \mathcal{C}$

```
while (y ◁ z)
  statement
```

Theorem

n, m: positive integers

$\mathbb{B}$: $\{0, 1\}$

task: compute a function mapping $\mathbb{B}^n$ to $\mathbb{B}^m$

Kernel C can be used to perform exactly the same tasks as your favourite programming language X++.

Fundamental theorem

P: some program

X: task performed by P

κ: # kernel-C statements in P

$T(n)$: running time of P for an input of size n

There exists a program P'

1. that has only one `while` statement,
2. that performs X,
3. whose length is $O(\kappa)$,
4. whose running time is $O(\kappa T(n))$.

For a proof, see, e.g. [Elmasry & Katajainen 2013]

Syntactic sugar

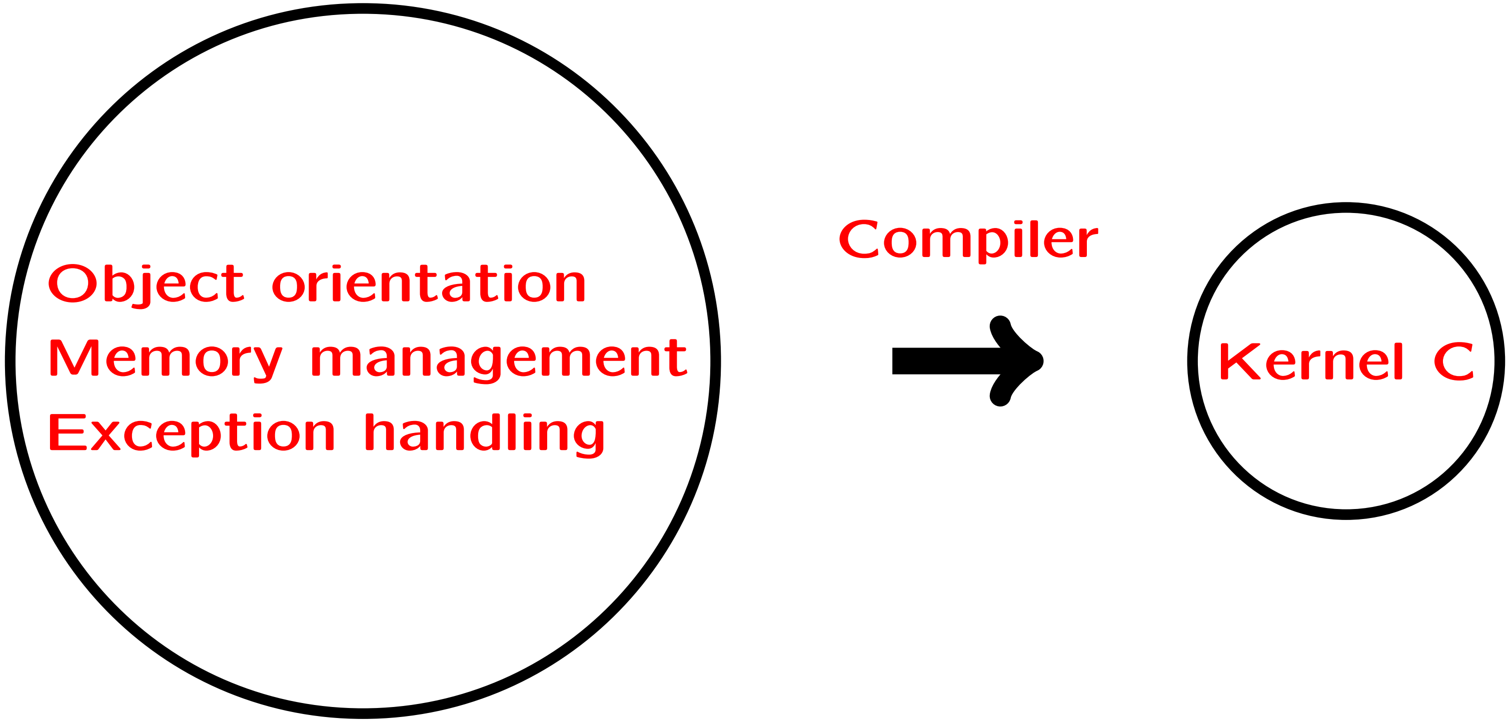
$$\begin{array}{l}
\texttt{if (expression)} \\
\quad \texttt{statement}_1 \\
\texttt{else} \\
\quad \texttt{statement}_2
\end{array}
\qquad
\begin{array}{l}
\texttt{done = 0;} \\
\texttt{if (expression)} \\
\quad \texttt{statement}_1 \\
\quad \texttt{done = 1;} \\
\texttt{if (! done \&\& ! expression)} \\
\quad \texttt{statement}_2
\end{array}$$

```
if (expression)
  statement
```

```
done = 0;
while (! done && expression)
  statement
  done = 1
```

$$\begin{array}{l}
\texttt{for (expression}_1\texttt{; expression}_2\texttt{; expression}_3\texttt{)} \\
\quad \texttt{statement}
\end{array}
\qquad
\begin{array}{l}
\texttt{expression}_1\texttt{;} \\
\texttt{while (expression}_2\texttt{)} \\
\quad \texttt{statement} \\
\quad \texttt{expression}_3\texttt{;}
\end{array}$$

Advanced features



Memory management

"Careful memory management—which includes releasing memory to prevent **memory leaks**—is crucial when developing kernel-level code."

```
/* allocate n bytes from the heap */
void* malloc(unsigned int n);

/* release the block pointed to by p */
void free(void* p);
```

For an implementation of a storage allocator, see [K & R, Section 8.7]

Kernel data structures

We expect that you know the following data structures (and know how to program them in C):

- singly-linked lists
- doubly-linked lists (Assignment 1)
- stack
- queue
- red-black tree
- hash tables
- bit vectors.

For a recap, read [Our textbook, Section 2.10]

Macros

Can be used

- to define constants,
- for conditional compilation, and
- to define macro functions.

```
#ifndef __bool_true_false_are_defined || __bool_true_false_are_defined == 0
  typedef int bool;
  #define false 0
  #define true !(false)
#endif
```

Inline assembly (gcc specific)

Basic inline assembly

```
asm("assembly code");
```

Example

```
asm("movl %ecx %eax");
```

Warning

The AT&T syntax used all over; information flows from left to right. In the above example the contents of `ecx` is moved to `eax`.

Extended inline assembly

```
asm( assembler template
  : output operands /* optional */
  : input operands /* optional */
  : list of clobbered  /* optional */
);
```

Example

```
int a = 10;
int b;
asm(
  "movl %1, %%eax\n"
  "movl %%eax, %0"
  : "=r" (b) /* output */
  : "r" (a) /* input */
  : "%eax" /* clobbered */
);
```

Why is good programming style important?

Clarity: Well-written code is easier to read and to understand.

Correctness: Sloppy code is often broken.

Simplicity: Well-written code is likely to be smaller than code that has been carelessly tossed together and never polished.

Maintainability: The best layout schemes hold up well under code modifications.

[Kernighan & Pike 1999, §1]

Idiomatic code

- Use expressions that are natural for a C programmer.
- Establish conventions in non-critical areas so that you can focus your creative energies in the places that count.
- Methods and tools that emphasize human discipline are especially effective. Form is liberating!

Document-comment idiom

Write a brief comment (if any) before an important function giving a high-level description what the function does, not how it does it.

```
/* count lines, words, and characters in input */
int main() {
  ...
  return 0;
}
```

Variable-naming conventions

- `c` is a character variable
- `i`, `j`, and `k` are integer indexes
- `n` is a number of something
- `p` and `q` are pointers
- `s` is a string
- Preprocessor macros are all in upper case as in `ALL_CAPS`
- Variable and routine names are all in lower case as in `all_lower_case`
- The underscore (_) character is used as a separator between words as above.

Variable-naming idioms

- Use descriptive names for global variables, and short names for local variables.
- Select sensible names for your variables. A misleading name can result in mysterious bugs.
- Use active names for functions. Also, functions that return a Boolean should be named so that the return value is unambiguous (e.g. `is_empty`).

Indentation idiom

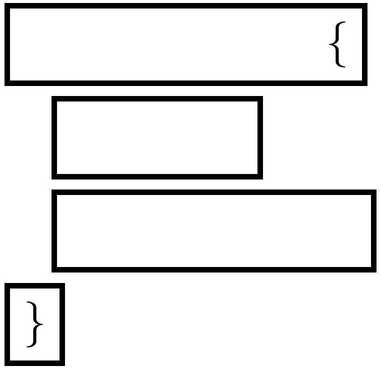
Indentation should be idiomatic, too. Use 2-4 space indentation to show the logical structure of a program. If you use tabs, make sure that your code-beautifier and other tools handle them appropriately.

```
  for (++n; n != 100; ++n) {
    field[n] = '\0';
  }
```

```
? for (
?   n++;
?   n != 100;
?   field[n++] = '\0';
? )
? {
?   ;
? }
```

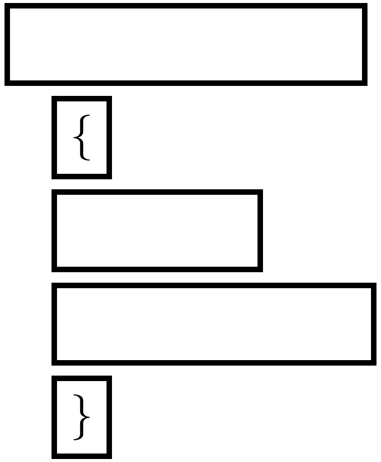
Which brace style is best?

Pure blocks:




```
while (colour == red) {
  statement1
  statement2
  ...
}
```

Braces as block boundaries:




```
while (colour == red)
  {
  statement1
  statement2
  ...
  }
```

Basic-loop idiom

Most loops, where the exit test is performed at the beginning, should have the form:

```
  for (i = 0; i != n; ++i) {
    ...
  }

  for (p = list; p != 0; p = (*p).next) {
    ...
  }

? i = 0;
? while (i <= n - 1) {
?   ...
? }

? for (i = 0; i < n; ) {
?   ...
?   i++;
? }

? for (i = n; --i >= 0; ) {
?   ...
? }
```

Infinite-loop idiom

For infinite loops use one of the following alternatives; do not use other forms.

```
  #include <stdbool.h>
  ...
  while (true) {
    ...
  }

  for (;;) {
    ...
  }

? while (1) {
?   ...
? }
```

End-exit idiom

A `do-while` loop always executes at least once; in many cases that behaviour is a bug waiting to bite, but when it is needed, write `}` in front of `while` to indicate clearly that `while` does not start a `while` loop.

```
safety_counter = 0;
do {
  current = (*current).next;
  ...
  ++safety_counter;
  if (safety_counter >= SAFETY_LIMIT) {
    fprintf(stderr, "Internal error: Safety-counter violation\n");
    assert(false);
  }
} while ((*current).next != 0);
```

Middle-exit idiom

Few programming languages provide direct support for a loop construct whose exit point is in the middle. An idiomatic way of writing such a loop is shown in the following example:

```
  while (true) {
    c = getchar();
    if (c == EOF) {
      break;
    }
    putchar();
  }

? while ((c = getchar()) != EOF) {
?   putchar();
? }
```

Switch-break idiom

Cases should almost always end with a **break**, with the rare exceptions commented. For an unusual structure, a sequence of **else-if** statements can be even clearer.

```
if (c == '-') {
  sign = -1;
  c = getchar();
}
else if (c == '+') {
  c = getchar();
}
else if (c != '.' && !isdigit(c)) {
  return 0;
}
```

```
? switch (c) {
? case '-': sign = -1;
? case '+': c = getchar();
? case '.': break;
? default:  if (!isdigit(c))
?               return 0;
? }
```

```
? switch (c) {
? case '-':
?   sign = -1;
?   /* fall through */
? case '+':
?   c = getchar();
?   break;
? case '.':
?   break;
? default:
?   if (!isdigit(c)) {
?     return 0;
?   }
?   break;
? }
```

Operation-per-line idiom

Avoid using multiple operations per line; in particular, be careful with statements that have side effects.

```
char* copy_string(char* s, char const* t) {
  char* const r = s;
  while (*t != '\0') {
    *s = *t;
    ++s;
    ++t;
  }
  *s = '\0';
  return r;
}
```

```
? void strcpy(char *s, char *t) {
?   while (*s++ = *t++)
?     ;
? }
```

Declaration-per-line idiom

Use only one data declaration per line.

```
FILE* input_file;
FILE* output_file;
```

```
? FILE *input_file, *output_file;
```

Separate the type names and variable names clearly in data declarations.

Memory-allocation idiom

In a real program, the return value of `malloc`, `realloc`, or any other allocation routine should always be checked.

```
  p = (char*) malloc(strlen(buffer) + 1);
  if (p == 0) {
    reset();
    return 0;
  }
  strcpy(p, buffer);

? p = malloc(strlen(buffer) + 1);
? strcpy(p, buffer);
```

Some comments on readability

Operators: Use spaces around operators: `x = y + z;`

,: Add a single space after commas: `a = f(x, y, z);`

!: Add a single space after `!`; negations are hard to understand and should be avoided.

if: Deeply nested `if` statements are difficult to follow.

->: Arrows clutter the code; my preference is `(*p).next`.

Instead of following the rules slavishly, it is more important to be consistent.

Some comments on maintainability

- Give names to magic numbers.
- Be prepared for changes.
- Use a commenting style that is easy to maintain.

```
#define SIZE(array) (sizeof(array) / sizeof(array[0])

char buffer[100];

for (i = 0; i != SIZE(buffer); ++i) {
  // Use braces even if the loop body only contains one statement.
}
```

Conclusion

C is quirky, flawed, and an enormous success.

[Ritchie 1993]

Critique of C

- Programmers have to do their own memory management—to declare variables, to manage pointer-chained lists, to dimension buffers, to detect and prevent buffer overruns, and to allocate and deallocate dynamic storage
- Little support for modularization
- No support for generic functions
- No string data type
- No support for variable-sized arrays
- Array syntax is a living fossil

```
int f(int a[]);
/* type of a is int* */
```

- Other syntax peculiarities

```
int* (*g)(int);
/* pointer to function
   returning int* */
```

- Semantic peculiarities

```
r = a >> 4 + 5;
/* shift right by 9 */
```

- Difficult to write numerical libraries.

Whence success

- Integration with the UNIX operating system
- A small and simple language
- Translatable with small and simple compilers
- Its types and operations are well-grounded in those provided by real machines
- Sufficiently abstract from machine details to achieve program portability
- Sufficiently expressive for describing time- and space-efficient programs
- Has remained remarkably stable
- Today, useful as a portable high-level assembly language.

Adopted from [Ritchie 1993]