U23 C Workshop

Florian “Florob” Zeitz

Chaos Computer Club Cologne e.V.
http://koeln.ccc.de

2013-10-19

Outline

- Initially written in the context of Unix

1978 “The C Programming Language” by Kernighan and Ritchie (first informal specification, K&R C)

1983 ANSI forms a committee to standardize C

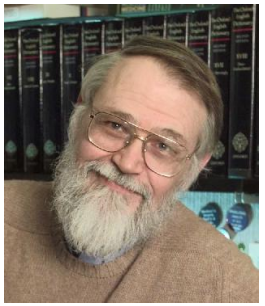
1988 “The C Programming Language” 2nd Edition, updated to reflect ANSI specification

1989 Specification approved by the ANSI (ANSI C/C89)

1990 Identical specification approved by the ISO (C90)

1999 Updated ISO specification (C99)

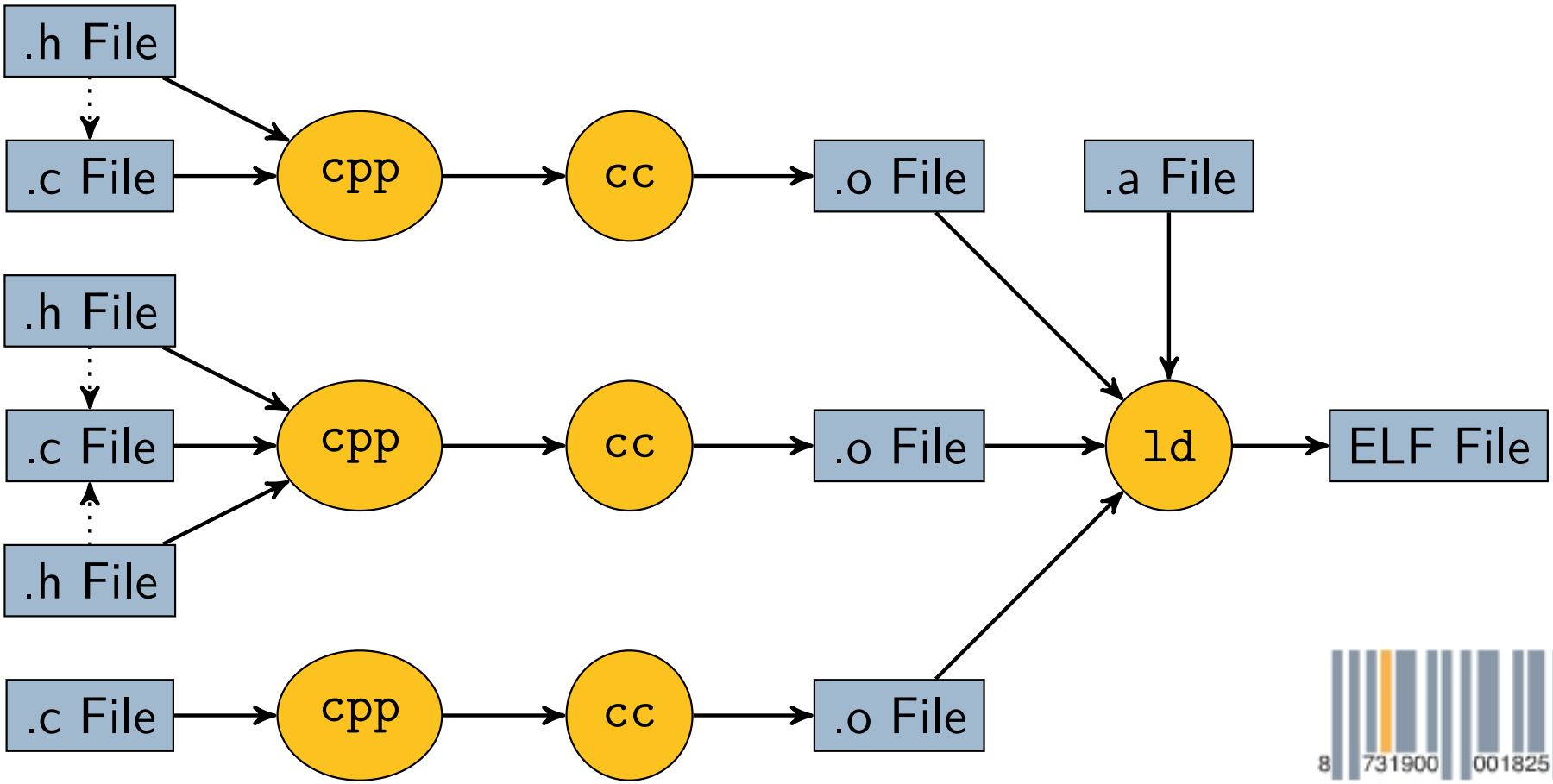
2011 Updated ISO specification (C11)



Brian Kernighan

Dennis Ritchie

Compiling code



Hello World

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello␣World\n");
6
7      return 0;
8  }
```

Hello World

```
1  #include <stdio.h> ← Include definitions for standard IO
2
3  int main(void)
4  {
5      printf("Hello World\n");
6
7      return 0;
8  }
```

Hello World

```
1  #include <stdio.h> ← Include definitions for standard IO
2
3  int main(void) ← Entry point
4  {
5      printf("Hello World\n");
6
7      return 0;
8  }
```

Hello World

```
1  #include <stdio.h> ← Include definitions for standard IO
2
3  int main(void) ← Entry point
4  {
5      printf("Hello World\n"); ← Write: Hello
                                  World<newline>
6
7      return 0;
8  }
```

Hello World

```
1  #include <stdio.h> ← Include definitions for standard IO
2
3  int main(void) ← Entry point
4  {
5      printf("Hello World\n"); ← Write: Hello
                                  World<newline>
6
7      return 0; ← Return success (0) to the
                   system
8  }
```

Expressions

- Expression: `printf("Hello␣Worldn")` ; — *Statement*
- "An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof."
- Almost everything is an expression
- An expression followed by a ; is a statement

Blocks

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello World\n");
6
7      return 0;
8  }
```

Block

- Also known as *compound statements*
- Collection of statements, often can be used in place of a single statement
- Relevant for scope (we'll talk about this later)

Type system

- C is statically typed
- Variables require a declaration, including type
- *type* `var;`
- Variables can be declared as `const` meaning their value can only be initialized, but never changed

Integer types

Name	Domain	Constant
`_Bool`	$\{0, 1\}$	0
`char`	$[-2^7, 2^7 - 1]$	5 or `'a'`
`int`	[INT_MIN, INT_MAX]	5
`unsigned int`	[0, UINT_MAX]	5u
`intX_t`	$[-2^{X-1}, 2^{(X-1)} - 1]$ $X \in \{8, 16, 32\}$	6
`uintX_t`	$[0, 2^X - 1]$ $X \in \{8, 16, 32\}$	6u

Floating-point types

Name	Domain	Constant
`float`	$\subset \mathbb{R}$	`1.5f` or `.3f` or `4.f` or `5e3f` or `0x4a.b2p4f`
`double`	$\subset \mathbb{R}$ (more values than `float`)	`1.5` or `.3` or `4.` or `5e3` or `0x4a.b2p4`

Void

- signals the absence of data (its domain is empty)
- `void` is an incomplete type

$\Rightarrow$ no variable of type `void` can be declared

Casts

- types can be converted to each other
- converting the value of an expression to another type is called cast
- this can be done explicitly by prefixing the expression by a type in parentheses
- e. g. `(uint8_t)1025` (effectively a modulo 256)

Scope

- region of program text where a variable is visible
- C uses file and block scope
- variables declared outside a block have file scope, others have block scope

Scope

```
1  void f(void)
2  {
3      int a;
4
5      {
6          int b;
7      }
8
9  }
```

Scope/Lifetime of b

Scope/Lifetime of a

Storage classes

- how variables are stored can be modified
- `auto`: lifetime is the associated block (default, rarely used explicitly)
- `static`: lifetime is the entire program execution
- `extern`: the variable belongs to another module
- `register`: access should be as fast as possible (in a register), can be ignored

printf()

```
int printf(const char *fmt, ...);
```

- takes a format string and any number of other parameters
- prints a string to stdout with the parameter formatted according to the format string

%i, %d prints an `int` (anything smaller than `int` is automatically converted to one here)

%f prints a `double` (`float`s are automatically converted to `double` here)

%s prints a `char*` (string)

%c prints an `int` as ASCII character

Escape sequences

\n	new line
\r	carriage return
\t	horizontal tab
\\	backslash
\'	single quote
\"	double quote
\<*oct*>	ASCII character <*oct*>
\x<*hex*>	ASCII character <*hex*>

printf()

```
1  int main(void) {
2    printf("%c: %i\n", 'a', 8);
3
4    return 0;
5  }
```

Output: `a: 8`

Boolean values

- Everything that is not equal to 0 is interpreted as true
- Everything equal to 0 is false
- Logical operations always evaluate to 0 or 1

Negation, Relational/Equality Operators

- `!a`: negation
- `a < b`: less than
- `a > b`: greater than
- `a <= b`: less than or equal
- `a >= b`: greater than or equal
- `a == b`: equal
- `a != b`: not equal

Logical AND/OR

- *exp1* && *exp2*: logical and (if *exp1* is false, *exp2* is not evaluated
- *exp1* || *exp2*: logical or (if *exp1* is true, *exp2* is not evaluated)
- left-to-right evaluation is guaranteed, side-effects of *exp2* might not take place

Arithmetic Operators

- `a + b`: Addition
- `a - b`: Subtraction
- `a * b`: Multiplication
- `a / b`: Division
- `a % b`: Modulo

Short forms

- `a += 3`: Same as `a = a + 3`
- `a -= 3`: Same as `a = a - 3`
- `a *= 3`: Same as `a = a * 3`
- `a /= 3`: Same as `a = a / 3`
- `a %= 3`: Same as `a = a % 3`
- `a++`: (Post-)Increment, evaluates to a's old value
- `a--`: (Post-)Decrement, evaluates to a's old value
- `++a`: (Pre-)Increment, evaluates to a's new value
- `--a`: (Pre-)Decrement, evaluates to a's new value

while-Loop

```
while(condition) statement/block
```

- Runs as long as *condition* is true
- *condition* is evaluated *before* each iteration

do-while-Loop

```
do statement/block while(condition)
```

- Runs as long as *condition* is true
- *condition* is evaluated *after* each iteration

$\Rightarrow$ runs at least once

for-Loop

```
for(initialization; condition; expression)
  statement/block
```

- Executes *initialization*
- Runs as long as *condition* is true
- *condition* is evaluated before each iteration
- Executes *Expression* after each iteration

Example: Print the alphabet

```
1 for (char c = 'a'; c <= 'z'; c = c+1)
2   putchar(c);
```

Changing the flow

- `continue`: Jumps imediately to the next loop iteration (checking the condition first)
- `break`: Terminates the loop prematurely

if-Statement

```
if(condition) statement/block
if(condition) statement/block else ↩
    statement/block
```

- if *condition* is true execute the first statement
- if *condition* is false execute the second statement

switch-Statement

`switch` (*condition*) *statement/block*

- jumps to a statement labeled "`case` *condition*" within the switch body
- if no such label exists jumps to a statement labeled "`default`"
- if no such label exists jumps past the switch body
- switch body can be left with `break`

Example: Fibonacci

```
1  int fib(int i)
2  {
3    switch(i) {
4    case 0:
5    case 1:
6      return i;
7    default:
8      return fib(i-1) + fib(i-2);
9    }
10 }
```

Example: Print a number

```
1  void printNumber(int num)
2  {
3     switch (num) {
4     case 0:
5        puts("Zero");
6        break;
7     case 1:
8        puts("One");
9        break;
10    default:
11       puts("Computers␣only␣use␣zeros␣and␣ones");
12    }
13 }
```

Exercises 1

Compiling code: `gcc -std=c99 -Wall -o output input.c`

1. Write, compile and execute a Hello World program
2. Write a program that prints the faculty of the numbers 0 to 10 using an iterative approach
3. Write a program that prints the first 10 fibonacci numbers using an iterative approach
4. Write a program that prints all primes between 2 and 100
5. Write a program that calculates the 5th power of all numbers from 2 to 10

Functions

- each function has a declaration and a definition
- declarations are usually provided in separate header files
- declaration: “This function exists and returns *type*”
- definition: “This function works as follows”

Declaration and Prototype

type1 `func(`*type2* `param1);`

- declares a function returning *type1*, with one parameter of type *type2*
- is both a declaration and a prototype
- prototype: “This function’s parameters have this types”
- parameter names may be omitted
- a function without parameters is declared with `void` as parameter list

Definition

type1 `func(`*type2* `param1)`
block

- defines a function
- must match the prototype
- can double as declaration/prototype

Exercises 2

1. Write a program that prints the faculty of the numbers 0 to 10 using a recursive approach
2. Write a program that prints the first 10 fibonacci numbers using a recursive approach
3. Implement a function that calculates the area of a triangle and test it
4. Implement a function returning the distance between two 3d points (6 double parameters) using `double sqrt(double x);` from `<math.h>`, add `-lm` to the compile command
5. Implement a program that counts the number of `'l'`s in the input using `int getchar(void);`

Bit Operations

- `a << b`: Shift `a` left by `b` Bit
- `a >> b`: Shift `a` right by `b` Bit
- `a & b`: Bitwise and
- `a | b`: Bitwise or
- `a ^ b`: Bitwise exclusive or
- These support the same short form as the arithmetic operations, e. g. `a ^= b`

Pointer type

- another scalar type
- points to another variable
- responsible for a lot of C's power
- also responsible for a lot of beginner confusion

Pointer type

- declared as *type* `*var`
- read “pointer to *type*”
- contains the address at which a variable is stored
- special value `NULL` to indicate that the pointer is not currently pointing anywhere

Address operator

```
1 int a;
2 int *a_p = &a;
```

- The `&` operator is used to get the address of a variable
- if `var` has the type *type* `&var` has the type *type*`*`
- above `a_p` is said to point to `a`

Indirection operator

```
1  int a;
2  int *a_p = &a;
3
4  *a_p = 5;
```

- The * operator is used to get the object stored at an address
- if var has the type *type** *var has the type *type*
- above *a_p = 5 sets the value of a to 5

`sizeof` operator

```
1  _Bool b;
2
3  if (sizeof(b) > sizeof(char))
4    printf("Booleans are rather large here\n");
```

- The `sizeof` operator determines the size of a variable or type
- The granularity is the length of a char (one byte)

Pointer arithmetic

```
1 int32_t a;
2 int32_t *a_p = &a;
3
4 a_p++;
```

- pointers store plain numbers (addresses)
- arithmetic works differently however
- addition and subtraction acts in the granularity of `sizeof(a)`
- E. g. a_p++ above increments the value of a_p by 4

Arrays

```
1  int A[4];
```

- aggregate data type
- contains a list of multiple adjacent variables
- `int A[4];` declares an array of 4 integers
- indexes start at 0

Accessing array members

```
1  int A[4];
2
3  *(A+2) = 3;
4  A[3] = 4;
```

- the expression `A` evaluates to a `int*` to the first element of `A`
- elements can therefore be accessed using pointer arithmetic
- e. g. `*(A+2)= 3` sets the 3rd element of `A` to 3
- `A[3]` is syntactic sugar for `(*((A)+(3)))`
- `3[A]` is therefore valid, but unintuitive

Structs

```
1  struct tag {
2    int i;
3    char c;
4  };
5  struct tag s;
6  struct tag *s_p = &s;
7
8  s.i = 5;
9  s_p->c = 'a';
```

- aggregate data type
- structured, composed of multiple variables of different types
- defined structures are referenced using a tag
- members are accessed using `.`
- `s_p->i` exists as syntactic sugar for `(*s_p).i`

Initializers

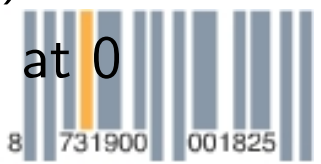
```
1 struct tag s = { 4, 'b' };
2 int A[4] = { 1, 2, 3, 4 };
3 struct tag t = { .c = 'd', .i = 4 };
4 int B[4] = { [2] = 3 };
5 char C[] = "Hello";
```

- aggregated types can be initialized using initializers
- if not otherwise specified members are initialized in order
- a designator can be given to address a specific member
- unnamed members are initialized to 0
- string literals can also be used as initializers
- array length can automatically be determined

Enums

```
1  enum tag {
2    NAME1, NAME2
3  };
4  enum month {
5    JAN = 1, FEB, MAR, APR, MAY, JUN,
6    JUL, AUG, SEP, OCT, NOV, DEZ
7  };
8  enum month birth_month;
```

- integer type with limited number of values
- other values can be assigned (acts like a normal integer)
- names are declared as integer constants, values starting at 0
- values can explicitly be assigned to a name



Compound Literals

```
1 struct tag s_p = &(struct tag){ 4, 5 };
```

- syntactically looks like casting a initializer
- defines an anonymous object
- scope and lifetime as if defined as a variable

Exercises 3

```c
char *fgets(char *s, int size, FILE *stream);
int atoi(const char *nptr); // From <stdlib.h>
```

1. Write a program that counts the number of 1 Bits in an integer read from stdin.
2. Write a function that exchanges the content of two integer variables
3. Write a function that sorts an integer array (nothing too fancy $\mathcal{O}(n^2)$ is fine)