

C Workshop

Kai, Florob

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

2016-11-05



Outline

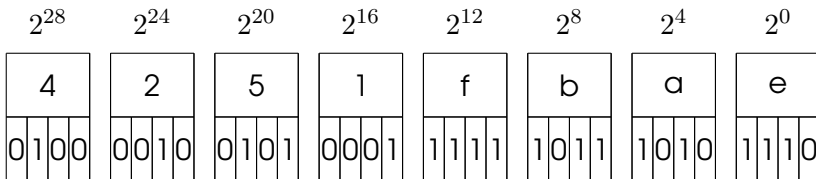
- 1 Hex
- 2 Hello u23
 - Hello World
 - Types
 - printf
- 3 Total Control
 - Arithmetic Operations
 - Logical Operations
 - Enums
 - Control Flow
 - Loops
- 4 Memory & Pointer
 - Pointer
 - Arrays
 - Memory Regions
 - Structs
 - C Strings
- 5 Getting Functional
 - Bit Operations
 - Functions
 - main()



- 1 Hex
- 2 Hello u23
 - Hello World
 - Types
 - printf
- 3 Total Control
 - Arithmetic Operations
 - Logical Operations
 - Enums
 - Control Flow
 - Loops
- 4 Memory & Pointer
 - Pointer
 - Arrays
 - Memory Regions
 - Structs
 - C Strings
- 5 Getting Functional
 - Bit Operations
 - Functions
 - main()



Hex



- each nibble represents 4 bit
- each digit increases its significance by factor $2^4 = 16$



- 1 Hex
- 2 Hello u23
 - Hello World
 - Types
 - printf
- 3 Total Control
 - Arithmetic Operations
 - Logical Operations
 - Enums
 - Control Flow
 - Loops
- 4 Memory & Pointer
 - Pointer
 - Arrays
 - Memory Regions
 - Structs
 - C Strings
- 5 Getting Functional
 - Bit Operations
 - Functions
 - main()



Why C?

- generated low level code is predictable
- more readable then assembly
- predominant language in systems programming



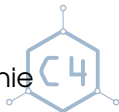
- Initially written in the context of Unix
- 1978 “The C Programming Language” by Kernighan and Ritchie (first informal specification, K&R C)
- 1983 ANSI forms a committee to standardize C
- 1988 “The C Programming Language” 2nd Edition, updated to reflect ANSI specification
- 1989 Specification approved by the ANSI (ANSI C/C89)
- 1990 Identical specification approved by the ISO (C90)
- 1999 Updated ISO specification (C99)
- 2011 Updated ISO specification (C11)



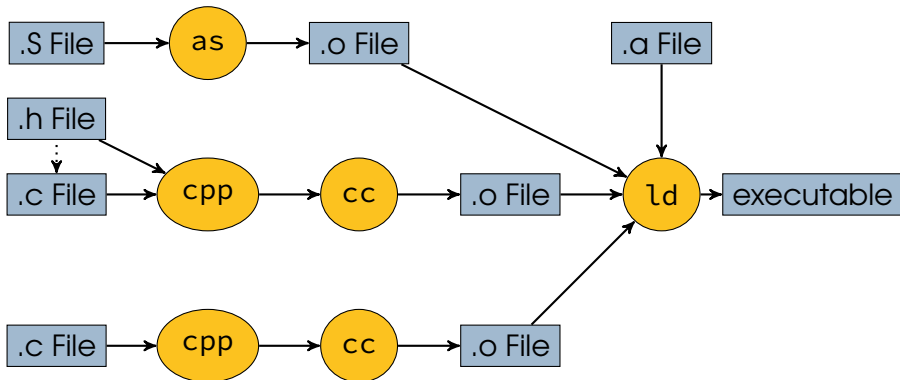
Brian Kernighan



Dennis Ritchie



Compiling code



Hello World

```
1  #include <stdio.h> ← 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 World\n ");
```

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

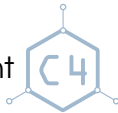


Blocks

```
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, 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'
unsigned char	$(0, 2^8)$	5u or 'a'
int	(INT_MIN, INT_MAX)	5
unsigned int	$(0, \text{UINT_MAX})$	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

hex **0x2a == 42**

octal **0622 == 402**



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
- ⇒ no variable of type **void** can be declared



Type conversion

- types can be converted between each other
- can happen implicitly as part of various operations
 - **3.1 + 4** results in a **double** with value **7.1**
 - **float** x = **4**; converts the **int** literal to a **float**
- explicitly converting the value of an expression to another type is called a *(type) cast*:
 - **(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



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 an **int** is automatically converted to one here)

%f prints a **double** (**floats** 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  {
3      printf("%c: %i\n", 'a', 8);
4
5      return 0;
6  }
```

Output: a: 8



Exercises

Compiling code: `gcc -std=c11 -Wall -o output input.c`

- 1 Write, compile and execute a Hello World program
- 2 Write a program that prints an **int**, **double**, and a **char** as a character
- 3 Write a program that prints the numbers from 1 to 3 and their squares in tabular form



Questions

- 1 How do you print a **float**?
- 2 What is the resulting type when subtracting **double** from **char**?
- 3 How large is **voids** domain?



Questions

```
1 int main(void)
2 {
3     int a = 4;
4     int b = 12;
5     {
6         a = a + 1;
7         b = a + b + 1;
8     }
```

```
9     {
10        int b = 5;
11        a = a + b + 3;
12        b = b + 4;
13    }
14    printf("a=%d, b=%d\n", a, b);
15    return 0;
16 }
```

- 4 What is the scope of each variable?
- 5 What is the output of this code? (no cheating)



- 1 Hex
- 2 Hello u23
 - Hello World
 - Types
 - printf
- 3 **Total Control**
 - Arithmetic Operations
 - Logical Operations
 - Enums
 - Control Flow
 - Loops
- 4 Memory & Pointer
 - Pointer
 - Arrays
 - Memory Regions
 - Structs
 - C Strings
- 5 Getting Functional
 - Bit Operations
 - Functions
 - main()



Arithmetic Operators

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



Short forms

`a = a + 3` \Rightarrow `a += 3`
`a = a - 3` \Rightarrow `a -= 3`
`a = a * 3` \Rightarrow `a *= 3`
`a = a / 3` \Rightarrow `a /= 3`
`a = a % 3` \Rightarrow `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



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

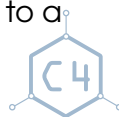
- $exp1 \ \&\& \ exp2$: logical and,
if $exp1$ is false, $exp2$ is not evaluated
- $exp1 \ || \ exp2$: logical or,
if $exp1$ is true, $exp2$ is not evaluated
- left-to-right evaluation is guaranteed
- side-effects of $exp2$ might not take place



Enums

```
1 enum tag {
2     NAME1, NAME2
3 };
4 enum month {
5     JAN = 1, FEB, MAR,
6     APR, MAY, JUN,
7     JUL, AUG, SEP,
8     OCT, NOV, DEZ
9 };
10 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



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



Labels & goto

```
1 void foo()
2 {
3     if (doA() == ERR)
4         goto exit;
5     if (doB() == ERR)
6         goto cleanupA;
7     if (doC() == ERR)
8         goto cleanupB;
9     /* everything has succeeded */
10    return;
11    cleanupB:
12        undoB();
13    cleanupA:
14        undoA();
15    exit:
16        return;
17 }
```

- labels can be declared by writing `ident:` at the beginning of a line
- `goto label` jumps to the statement at that label



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



while-Loop

while (*condition*) *statement/block*

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

```
1 while (a > 5)
2   a /= 2;
```



do-while-Loop

do *statement/block* **while** (*condition*);

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

⇒ runs at least once

```
1 do {  
2     b++;  
3 } while (b < 10);
```



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

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



Changing the flow

- **continue**: Jumps immediately to the next loop iteration, terminates if *condition* became false
- **break**: Terminates the loop prematurely



Exercises

- 1 Write a program that prints the first 10 Fibonacci numbers (iterative)
- 2 Write a program that prints all prime numbers between 2 and 100
- 3 Implement FizzBuzz:
 - Iterate over all numbers from 1 to 100
 - Print "Fizz" if the number is divisible by 3
 - Print "Buzz" if the number is divisible by 5
 - Print "FizzBuzz" if the number is divisible by 3 and 5
 - Print the number if none of the above apply



Questions

- 1 How do you rewrite a **while**-Loop, as a **for**-Loop?
- 2 What are **enums** useful for?



Questions

```
1  int main(void)
2  {
3      int a = 0;
4      int b = 0;
5      int c = ++a + b++;
6      b += ++a + c;
7      a += b++ + c++;
8      printf("a=%d, b=%d, c=%d\n", a, b, c);
9
10     return 0;
11 }
```

- ③ What is the output of the above code? (no cheating)



- 1 Hex
- 2 Hello u23
 - Hello World
 - Types
 - printf
- 3 Total Control
 - Arithmetic Operations
 - Logical Operations
 - Enums
 - Control Flow
 - Loops
- 4 Memory & Pointer
 - Pointer
 - Arrays
 - Memory Regions
 - Structs
 - C Strings
- 5 Getting Functional
 - Bit Operations
 - Functions
 - main()



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

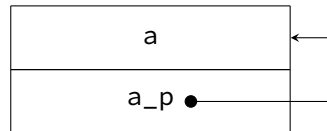
```

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

```

0xbfdd193c:

0xbfdd1940:



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

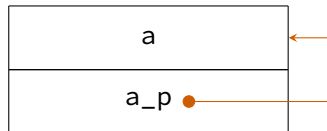


Indirection operator

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

0xbf8f825c:

0xbf8f8260:



- 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\n");
```

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



Arrays

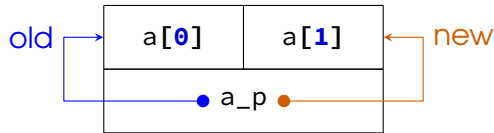
```
1 int A[4];
```

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



Pointer arithmetic

```
1 int16_t a[2];
2 int16_t *a_p = a;
3
4 a_p++;
```



- pointers store plain numbers (addresses)
- arithmetic works differently
- addition and subtraction acts in the granularity of `sizeof(*a)`
- e. g. `a_p++` above increments the value of `a_p` by 2



Accessing array members

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

- the expression A evaluates to an **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

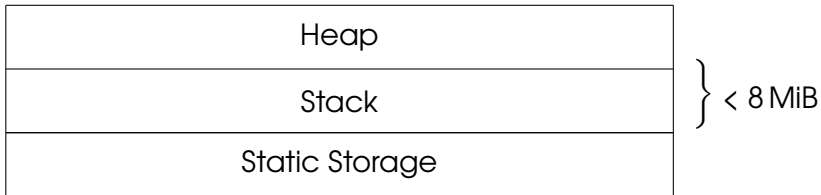


Storage classes

- 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



Memory regions



Stack local variables, relatively small

Heap large data, data live across functions

Static global variables, variables declared **static**



malloc()

```
1 void *malloc(size_t size);  
2 void *calloc(size_t nmemb, size_t size);  
3 void free(void *ptr);
```

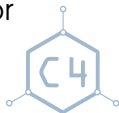
- used to allocate storage on the heap
- failure is indicated by returning `NULL`
- `calloc()` zeros the memory
- `free()` deallocates memory



Structs

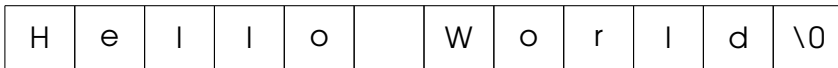
```
1 struct tag {
2     int i;
3     char c;
4 };
5 struct tag s;
6 struct tag *s_p = &s;
7
8 s.i = 5;
9 s_p->c = 'a';
```

- 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`



C Strings

```
1 char hello[] = "Hello World";
```



- represented as array of **char**
- terminated by a zero byte
- literal has type **char***



Exercises

- 1 Determine the size of a `_Bool`, `int`, `double`, and `char`, using `sizeof` (`%zu` prints a `size_t`)



Exercises

```
1 enum product {
2     TOOTHPASTE, // 1,70€
3     GINGER,     // 0,90€
4     RICE,       // 2,30€
5     TOMATO      // 0,60€
6 }
```

```
1 struct entry {
2     enum product prod;
3     unsigned int num;
4 }
```

- 2 allocate and initialize a grocery list with 10 entries on the heap
- 3 calculate the total price of all items on the list



Questions

- 1 Why does the Heap exist?
- 2 Which memory region contains variables declared with function scope?
- 3 Which memory region contains variables declared outside a function?



- 1 Hex
- 2 Hello u23
 - Hello World
 - Types
 - printf
- 3 Total Control
 - Arithmetic Operations
 - Logical Operations
 - Enums
 - Control Flow
 - Loops
- 4 Memory & Pointer
 - Pointer
 - Arrays
 - Memory Regions
 - Structs
 - C Strings
- 5 Getting Functional
 - Bit Operations
 - Functions
 - main()



Bit Operations

- $a \ll b$: Shift a left by b bit
- $a \gg b$: Shift a right by b bit
- $a \& b$: Bitwise and
- $a | b$: Bitwise or
- $a \wedge b$: Bitwise exclusive or
- These support the same short form as the arithmetic operations, e.g. $a \wedge= b$



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



Prototype

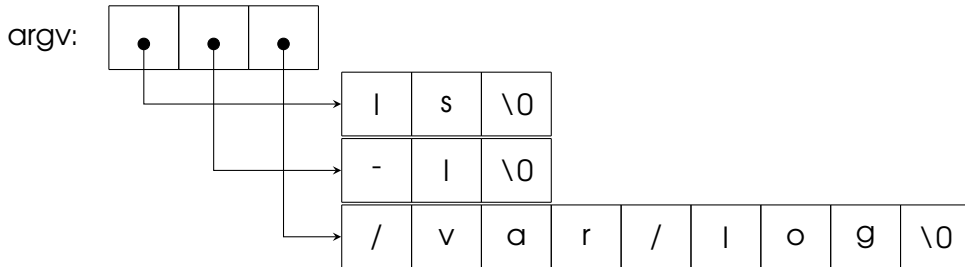
```
1 int main(void);  
2 int main(int argc, char **argv);  
3 int main(int argc, char *argv[]); // equivalent to line 2  
4 int main(int argc, char *argv[], char *envp[]); // non-standard
```

- argc: number of arguments
- argv: array of argument strings
- envp: array of environment strings, **NULL** terminated



main()

argv/envp



Return value

- returned to the invoking process/system
- 0 signals success, everything else signals an error



main()

C is not a macro assembler

```
1  int main(void)
2  {
3      int a, b;
4
5      if (a) {
6          b = 3;
7      } else {
8          a = 4;
9      }
10     printf("%i\n", a + b);
11
12     return 0;
13 }
```



Exercises

- 1 Write a program that computes the n th Fibonacci number recursively, taking n from `argv` (man 3 atoi)
- 2 Write a function realizing a rotate left on **unsigned int**
- 3 Write a program counting the **1** bits in `atoi(argv[1])`
- 4 Write a function that swaps the contents of two integer variables



Questions

- 1 What does `argv[0]` contain?
- 2 What is `envp`?
- 3 What's the result of right shifting a signed integer?
- 4 What is the return value of `main()` used for?

