C Workshop

Florob

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

Saturday 14th October, 2017

Outline

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Hex

2^{28}	2^{24}	2^{20}	2^{16}	2^{12}	2^{8}	2^{4}	2^{0}
4	2	5	1	f	b	a	e
0100	0010	0101	0001	1111	1011	1010	1110

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

Why C?

- generated low level code is predictable (timing, power consumption)
- predominant language in embedded programming
- libraries available

- 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

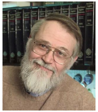
1988 "The C Programming Language" 2nd Edition, updated to reflect ANSI specification

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

1990 Identical specification approved by the ISO (C90)

1999 Updated ISO specification (C99)

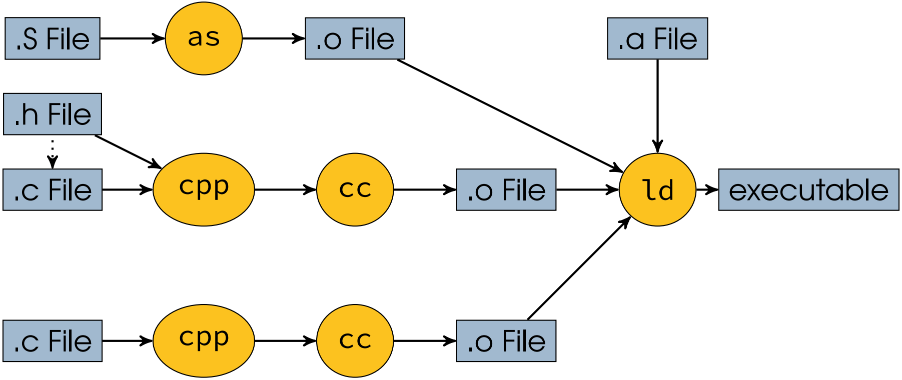
2011 Updated ISO specification (C11)



Brian Kernighan

Dennis Ritchie

Compiling code



Hello World

```
#include <stdio.h> ← Include definitions for standard IO

int main(void) ← Entry point
{
  printf("Hello World\n"); ← Write: Hello World<newline>

  return 0; ← Return success (0) to the system
}
```


Florian “Florob” Zeitz <florob@babelmonkeys.de>

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Blocks

```c
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");

  return 0;
}
```

Block (lines 4–8, `{` … `}`)

- 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)

Florian "Florob" Zeitz <florob@babelmonkeys.de>

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, `UINT_MAX`)	`5u`
`int`*X*`_t`	$(-2^{X-1}, 2^{(X-1)} - 1); X \in \{8, 16, 32\}$	`6`
`uint`*X*`_t`	$(0, 2^X - 1); X \in \{8, 16, 32\}$	`6u`

hex `0x2a == 42`

octal `0622 == 402`

Florian “Florob” Zeitz <florob@babelmonkeys.de>

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**

Florian "Florob" Zeitz <florob@babelmonkeys.de>

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Scope

```
void f(void)
{
   int a;

   {
      int b;
   }

}
```

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` (`float`s are automatically converted to `double` here)

%s prints a `char*` (string)

%c prints an `int` as ASCII character

Florian "Florob" Zeitz <florob@babelmonkeys.de>

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*>

Florian "Florob" Zeitz <florob@babelmonkeys.de>

printf()

```c
int main(void)
{
  printf("%c: %i\n", 'a', 8);

  return 0;
}
```

Output: `a: 8`

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Exercises

Compiling code: `gcc -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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Questions

1. How do you print a `float`?
2. What is the resulting type when subtracting `double` from `char`?
3. How large is `void`s domain?

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Questions

```c
int main(void)
{
  int a = 4;
  int b = 12;
  {
    a = a + 1;
    b = a + b + 1;
  }
  {
    int b = 5;
    a = a + b + 3;
    b = b + 4;
  }
  printf("a=%d, b=%d\n", a, b);
  return 0;
}
```

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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Arithmetic Operators

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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Short forms

```
a = a + 3  ⇒  a += 3
a = a - 3  ⇒  a -= 3
a = a * 3  ⇒  a *= 3
a = a / 3  ⇒  a /= 3
a = a % 3  ⇒  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`

Florian “Florob” Zeitz <florob@babelmonkeys.de>

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Enums

```c
enum tag {
   NAME1, NAME2
};
enum month {
   JAN = 1, FEB, MAR,
   APR, MAY, JUN,
   JUL, AUG, SEP,
   OCT, NOV, DEZ
};
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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

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`

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Example: Fibonacci

```
unsigned int fib(unsigned int i)
{
  switch (i) {
  case 0:
  case 1:
    return i;
  default:
    return fib(i-1) + fib(i-2);
  }
}
```


Florian “Florob” Zeitz <florob@babelmonkeys.de>

Example: Print a number

```
void printNumber(int num) {
  switch (num) {
  case 0:
    puts("Zero");
    break;
  case 1:
    puts("One");
    break;
  default:
    puts("Computers only use zeros and ones");
  }
}
```


Florian “Florob” Zeitz <florob@babelmonkeys.de>

while-Loop

while (*condition*) *statement/block*

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

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


Florian "Florob" Zeitz <florob@babelmonkeys.de>

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

```c
do {
   b++;
} while (b < 10);
```


Florian “Florob” Zeitz <florob@babelmonkeys.de>

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

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


Florian "Florob" Zeitz <florob@babelmonkeys.de>

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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Questions

1. How do you rewrite a `while`-Loop, as a `for`-Loop?
2. What are `enum`s useful for?

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Questions

```c
int main(void)
{
        int a = 0;
        int b = 0;
        int c = ++a + b++;
        b += ++a + c;
        a += b++ + c++;
        printf("a=%d, b=%d, c=%d\n", a, b, c);

        return 0;
}
```

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

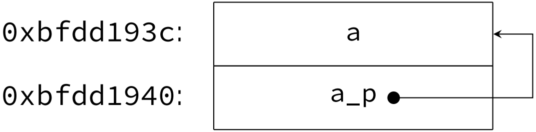
Pointer type

- 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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Address operator

```c
int a;
int *a_p = &a;
```



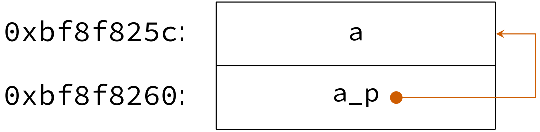
- The & operator is used to get the address of a variable
- if var has the type **type**,
  then &var has the type **type***
- above a_p is said to point to a

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Indirection operator

```c
int a;
int *a_p = &a;

*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`

Florian "Florob" Zeitz <florob@babelmonkeys.de>

sizeof operator

```c
_Bool b;

if (sizeof(b) > sizeof(char))
  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)

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Arrays

```c
int A[4];
```

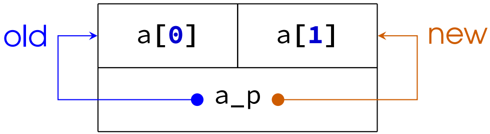
- aggregate data type
- contains a list of multiple adjacent variables
- `int A[4];` declares an array of 4 integers
- indexes start at `0`

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Pointer arithmetic

```c
int16_t a[2];
int16_t *a_p = a;

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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Accessing array members

```c
int A[4];

*(A+2) = 3;
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

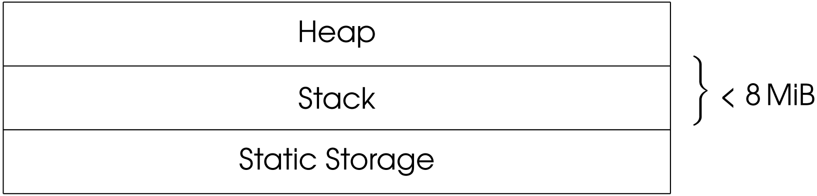
Florian "Florob" Zeitz <florob@babelmonkeys.de>

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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

Memory regions



Stack	local variables, relatively small
Heap	large data, data live across functions
Static	global variables, variables declared `static`

Florian “Florob” Zeitz <florob@babelmonkeys.de>

malloc()

```
#include <stdio.h>
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
```

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Structs

```c
struct tag {
  int i;
  char c;
};
struct tag s;
struct tag *s_p = &s;

s.i = 5;
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`

Florian "Florob" Zeitz <florob@babelmonkeys.de>

C Strings

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

H	e	l	l	o		W	o	r	l	d	\0

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Exercises

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

Exercises

```c
enum product {
  TOOTHPASTE, // 1,70€
  GINGER,     // 0,90€
  RICE,       // 2,30€
  TOMATO      // 0,60€
}
```

```c
struct entry {
  enum product prod;
  unsigned int num;
}
```

2. allocate and initialize a grocery list with 5 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?

Florian "Florob" Zeitz <florob@babelmonkeys.de>

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`

Florian "Florob" Zeitz <florob@babelmonkeys.de>

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

Florian "Florob" Zeitz <florob@babelmonkeys.de>

Definition

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

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

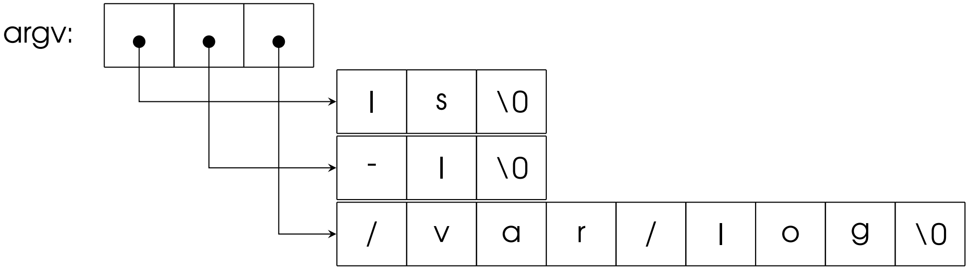
Prototype

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

- `argc`: number of arguments
- `argv`: array of argument strings

Florian "Florob" Zeitz <florob@babelmonkeys.de>

argv



Florian "Florob" Zeitz <florob@babelmonkeys.de>

Return value

- returned to the invoking process/system
- 0 signals success, everything else signals an error
- usually irrelevant on embedded systems

Compound Literals

```c
struct tag *s_p = &(struct tag){ 4, 5 };
int *x = (int[3]){ 0x2a, 42, 7 };
```

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

C is not a macro assembler

```c
int main(void)
{
  int a, b;

  if (a) {
    b = 3;
  } else {
    a = 4;
  }
  printf("%i\n", a + b);

  return 0;
}
```


Florian “Florob” Zeitz <florob@babelmonkeys.de>

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

Florian “Florob” Zeitz <florob@babelmonkeys.de>

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?

Florian “Florob” Zeitz <florob@babelmonkeys.de>