What you thought you knew about C

Florian “Florob” Zeitz

2015-03-25

Disclaimer: What are we talking about?

- C99 and/or C11
- not necessarily C++
- but Objective-C, as it works as a real superset

Disclaimer: What are we talking about?

- C99 and/or C11
- not necessarily C++
- but Objective-C, as it works as a real superset

Disclaimer: What are we talking about?

- C99 and/or C11
- not necessarily C++
- but Objective-C, as it works as a real superset

1. Undefined behaviour
2. Strict aliasing
3. Arrays
4. Conversions
5. Fun with C99 (and above)

1. Undefined behaviour
2. Strict aliasing
3. Arrays
4. Conversions
5. Fun with C99 (and above)

What’s this?

Multiple types of “behaviour”:

implementation-defined behaviour

documented implementation choice (e. g. signedness of `char`)

unspecified behaviour

more than one possibility (e. g. evaluation of function arguments)

undefined behaviour

everything goes, input program is considered erroneous (e. g. use-after-free)

What’s this?

Multiple types of “behaviour”:

implementation-defined behaviour

documented implementation choice (e. g. signedness of `char`)

unspecified behaviour

more than one possibility (e. g. evaluation of function arguments)

undefined behaviour

everything goes, input program is considered erroneous (e. g. use-after-free)

What’s this?

Multiple types of “behaviour”:

implementation-defined behaviour

documented implementation choice (e. g. signedness of `char`)

unspecified behaviour

more than one possibility (e. g. evaluation of function arguments)

undefined behaviour

everything goes, input program is considered erroneous (e. g. use-after-free)

What does this snippet usually print?
(Unoptimized, on a x86 system)

```c
uint32_t shifty = 1;
shifty = shifty << 32;
printf(”%”PRIu32”\n”, shifty);
```

0

1

42

neither

What does this snippet usually print?
(Unoptimized, on a x86 system)

```c
uint32_t shifty = 1;
shifty = shifty << 32;
printf("%"PRIu32"\n", shifty);
```

0 [✓ 1]

42 neither

Oversized shift amounts

> *If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.*

- set variables to zero instead
- easily checked when type width is known

Oversized shift amounts

> *If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.*

- set variables to zero instead
- easily checked when type width is known

Oversized shift amounts

> *If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.*

- set variables to zero instead
- easily checked when type width is known

What does this snippet usually print when size is INT_MAX?
(Optimized with -O3)

```
int size = ...;
if (size > size+1) {
  puts(”Aborted”)
  abort();
}
puts(”Fetching␣memory”);
malloc(size+1);
```

Nothing

”Aborted”

”Fetching memory”

size

What does this snippet usually print when size is INT_MAX? (Optimized with -O3)

```
int size = ...;
if (size > size+1) {
  puts(”Aborted”)
  abort();
}
puts(”Fetching␣memory”);
malloc(size+1);
```

Nothing

”Aborted”

[✓ ”Fetching memory”]

size

Signed integer overflow

- unsigned integer overflow is well-defined: `UINT_MAX+1 = 0`
- signed integer overflow is not: `INT_MAX+1 =` *undef*
- rumours aside `INT_MAX+1` is not `INT_MIN`
- Check equality against `INT_MAX`

Signed integer overflow

- unsigned integer overflow is well-defined: `UINT_MAX+1 = 0`
- signed integer overflow is not: `INT_MAX+1 = ` *undef*
- rumours aside `INT_MAX+1` is not `INT_MIN`
- Check equality against `INT_MAX`

Signed integer overflow

- unsigned integer overflow is well-defined: `UINT_MAX+1 = 0`
- signed integer overflow is not: `INT_MAX+1 =` *undef*
- rumours aside `INT_MAX+1` is not `INT_MIN`
- Check equality against `INT_MAX`

Signed integer overflow

- unsigned integer overflow is well-defined: `UINT_MAX+1 = 0`
- signed integer overflow is not: `INT_MAX+1 =` *undef*
- rumours aside `INT_MAX+1` is not `INT_MIN`
- Check equality against `INT_MAX`

```
int size = ...;
if (size > size+1) {
  puts(”Aborted”)
  abort();
}
puts(”Fetching␣memory”);
malloc(size+1);
```

- Only defined behavior is considered
- `size > size+1` is always false
- Optimization removes the branch

```c
int size = ...;
if (size > size+1) {
  puts(”Aborted”)
  abort();
}
puts(”Fetching␣memory”);
malloc(size+1);
```

- Only defined behavior is considered
- `size > size+1` is always false
- Optimization removes the branch

```
int size = ...;
if (size > size+1) {
  puts(”Aborted”)
  abort();
}
puts(”Fetching␣memory”);
malloc(size+1);
```

- Only defined behavior is considered
- `size > size+1` is always false
- Optimization removes the branch

What you already knew

There are other well-known examples:

- Dereferencing NULL pointers
- Dereferencing wild pointers
- Out-of-bound array indices
- Use-after-free

What you already knew

There are other well-known examples:

- Dereferencing NULL pointers
- Dereferencing wild pointers
- Out-of-bound array indices
- Use-after-free

What you already knew

There are other well-known examples:

- Dereferencing NULL pointers
- Dereferencing wild pointers
- Out-of-bound array indices
- Use-after-free

What you already knew

There are other well-known examples:

- Dereferencing NULL pointers
- Dereferencing wild pointers
- Out-of-bound array indices
- Use-after-free

Mitigation

- compiler warnings: `-Wall`
- runtime checks: `-ftrapv`, `-fsanitize=undefined` and friends
- make signed overflow wrap: `-fwrapv`
- static analyzers: e. g. Clang Static Analyzer, (sp)lint
- dynamic analyzers: e. g. Valgrind

Mitigation

- compiler warnings: `-Wall`
- runtime checks: `-ftrapv`, `-fsanitize=undefined` and friends
- make signed overflow wrap: `-fwrapv`
- static analyzers: e. g. Clang Static Analyzer, (sp)lint
- dynamic analyzers: e. g. Valgrind

Mitigation

- compiler warnings: `-Wall`
- runtime checks: `-ftrapv`, `-fsanitize=undefined` and friends
- make signed overflow wrap: `-fwrapv`
- static analyzers: e. g. Clang Static Analyzer, (sp)lint
- dynamic analyzers: e. g. Valgrind

Mitigation

- compiler warnings: `-Wall`
- runtime checks: `-ftrapv`, `-fsanitize=undefined` and friends
- make signed overflow wrap: `-fwrapv`
- static analyzers: e. g. Clang Static Analyzer, (sp)lint
- dynamic analyzers: e. g. Valgrind

Mitigation

- compiler warnings: `-Wall`
- runtime checks: `-ftrapv`, `-fsanitize=undefined` and friends
- make signed overflow wrap: `-fwrapv`
- static analyzers: e. g. Clang Static Analyzer, (sp)lint
- dynamic analyzers: e. g. Valgrind

1. Undefined behaviour
2. Strict aliasing
3. Arrays
4. Conversions
5. Fun with C99 (and above)

What does this snippet usually print?
(Optimized, clang or gcc)

```c
void f(int *i, float *f) {
  *i = 42;
  *f = 13;
  printf(”%i\n”, *i);
}
```

```c
int main(void) {
  int var;
  f(&var,  &var);
  return 0;
}
```

42

13

0

1095761920

What does this snippet usually print?
(Optimized, clang or gcc)

```c
void f(int *i, float *f) {
  *i = 42;
  *f = 13;
  printf(”%i\n”, *i);
}
```

```c
int main(void) {
  int var;
  f(&var,  &var);
  return 0;
}
```

[✓ 42]

13

0

1095761920

- C allows aliasing
- `int *pa = &a`, `*pa` aliases `a`
- not all expressions may be used to access an object
- expression and object type must match
- this restriction is commonly called the *strict aliasing rule*
- with `a` declared as a `float`, `*pa` may be neither read nor written

- C allows aliasing
- `int *pa = &a`, `*pa` aliases `a`
- not all expressions may be used to access an object
- expression and object type must match
- this restriction is commonly called the *strict aliasing rule*
- with `a` declared as a `float`, `*pa` may be neither read nor written

- C allows aliasing
- `int *pa = &a`, *pa aliases a
- not all expressions may be used to access an object
- expression and object type must match
- this restriction is commonly called the *strict aliasing rule*
- with a declared as a `float`, *pa may be neither read nor written

- C allows aliasing
- `int *pa = &a`, *pa aliases a
- not all expressions may be used to access an object
- expression and object type must match
- this restriction is commonly called the *strict aliasing rule*
- with a declared as a `float`, *pa may be neither read nor written

- C allows aliasing
- `int *pa = &a`, `*pa` aliases `a`
- not all expressions may be used to access an object
- expression and object type must match
- this restriction is commonly called the *strict aliasing rule*
- with `a` declared as a `float`, `*pa` may be neither read nor written

- C allows aliasing
- **int** *pa = &a, *pa aliases a
- not all expressions may be used to access an object
- expression and object type must match
- this restriction is commonly called the *strict aliasing rule*
- with a declared as a **float**, *pa may be neither read nor written

Exceptions

- different signedness
- different qualifiers
- struct, array or union type with a member of one of the aforementioned types
- character type

Exceptions

- different signedness
- different qualifiers
- struct, array or union type with a member of one of the aforementioned types
- character type

Exceptions

- different signedness
- different qualifiers
- struct, array or union type with a member of one of the aforementioned types
- character type

Exceptions

- different signedness
- different qualifiers
- struct, array or union type with a member of one of the aforementioned types
- character type

```c
void func(int *i, float *f) {
  *i = 5;
  *f = 42.0f;
  g(*i);
}
```

- potential for constant propagation
- if aliasing is desired, the object needs to be in a union with all types

```c
void func(int *i, float *f) {
  *i = 5;
  *f = 42.0f;
  g(*i);
}
```

- potential for constant propagation
- if aliasing is desired, the object needs to be in a union with all types

1. Undefined behaviour
2. Strict aliasing
3. Arrays
4. Conversions
5. Fun with C99 (and above)

Consider the declaration `char A[2]`

What is the type of this expression?

```
A
```

array of char

char

int

pointer to char

Consider the declaration `char A[2]`
What is the type of this expression?

```
A
```

array of char

char

int

[✓ pointer to char]

Consider the declaration `char B[3][5]`
What is the type of this expression?

B

array of array of char

pointer to array of char

array of pointer to char

pointer to pointer to char

Consider the declaration `char B[3][5]`
What is the type of this expression?

B

array of array of char

[✓ pointer to array of char]

array of pointer to char

pointer to pointer to char

How do you declare a pointer to an array (3) of int?

```
int (*C)[3]
```

```
int *C[3]
```

```
int C[][3]
```

```
int &C[3]
```

How do you declare a pointer to an array (3) of int?

[✓ `int (*C)[3]`]

`int *C[3]`

`int C[][3]`

`int &C[3]`

Arrays

- generally well-understood
- confusion about their relation to pointers

  *Except when it is the operand of the **`sizeof`** operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted to an expression with type "pointer to type" that points to the initial element of the array object and is not an lvalue.*

Arrays

- generally well-understood
- confusion about their relation to pointers

> *Except when it is the operand of the* **`sizeof`** *operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type “array of type” is converted to an expression with type “pointer to type” that points to the initial element of the array object and is not an lvalue.*

Arrays

- generally well-understood
- confusion about their relation to pointers

Except when it is the operand of the ***`sizeof`*** *operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted to an expression with type "pointer to type" that points to the initial element of the array object and is not an lvalue.*

So it is basically a **char****, right?

```
char B[3][5] = {”Word”, ”CCCC”, ”axes”};
char **Bp = B;
```

- `warning: initialization from incompatible pointer type`
- “pointer to array of char” and “pointer to pointer to char” is not the same thing
- `char (*Bp)[5]`

So it is basically a **char****, right?

```
char B[3][5] = {”Word”, ”CCCC”, ”axes”};
char **Bp = B;
```

- `warning: initialization from incompatible pointer type`
- “pointer to array of char” and “pointer to pointer to char” is not the same thing
- `char (*Bp)[5]`

So it is basically a char**, right?

```
char B[3][5] = {”Word”, ”CCCC”, ”axes”};
char **Bp = B;
```

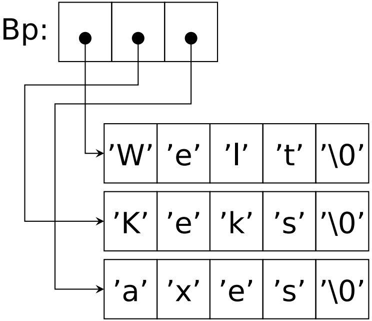
- `warning: initialization from incompatible pointer type`
- “pointer to array of char” and “pointer to pointer to char” is not the same thing
- `char (*Bp)[5]`

And if I just cast it?

```c
char B[3][5] = {”Welt”, ”Keks”, ”axes”};
char **Bp = (char*[3]){”Welt”, ”Keks”, ”axes”};
```

B:

’W’	’e’	’l’	’t’	’\0’
’K’	’e’	’k’	’s’	’\0’
’a’	’x’	’e’	’s’	’\0’



’W’	’e’	’l’	’t’	’\0’
’K’	’e’	’k’	’s’	’\0’
’a’	’x’	’e’	’s’	’\0’

Consider the declaration `char B[3][5]`
How many Bytes after the start of B does the following expression read?
`B + 1`

1

5

10

3

Consider the declaration `char B[3][5]`
How many Bytes after the start of B does the following expression read?

```
B + 1
```

1

[✓ 5]

10

3

1. Undefined behaviour
2. Strict aliasing
3. Arrays
4. Conversions
5. Fun with C99 (and above)

What does this snippet usually print? (Optimized, clang or gcc)

```c
signed int s = -1;
unsigned int u = 1;
if (s < u)
  puts(”True”);
else
  puts(”False”);
```

”True”

Nothing

”False”

”Trlse”

What does this snippet usually print? (Optimized, clang or gcc)

```c
signed int s = -1;
unsigned int u = 1;
if (s < u)
  puts(”True”);
else
  puts(”False”);
```

”True”

Nothing

[✓ ”False”]

”Trlse”

What does this snippet print?

```
unsigned int u = 1;
signed int s1 = -2;
signed int s2 = u + s1;
printf(”%i\n”, s2);
```

-1

0

4294967295

1

What does this snippet print?

```c
unsigned int u = 1;
signed int s1 = -2;
signed int s2 = u + s1;
printf(”%i\n”, s2);
```

[✓ -1] 0

4294967295 1

Integer conversion ranks

- Used to determine which integer type to convert to
- No fixed mapping
- Roughly: larger range of values $\Rightarrow$ higher rank

Integer promotions

- Only applied to expressions with integer type of rank lower than `(unsigned)int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Integer promotions

- Only applied to expressions with integer type of rank lower than (`unsigned`)`int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Integer promotions

- Only applied to expressions with integer type of rank lower than `(unsigned)int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Integer promotions

- Only applied to expressions with integer type of rank lower than (`unsigned`)`int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Integer promotions

- Only applied to expressions with integer type of rank lower than (`unsigned`)`int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Integer promotions

- Only applied to expressions with integer type of rank lower than (`unsigned`)`int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Integer promotions

- Only applied to expressions with integer type of rank lower than `(unsigned)int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Integer promotions

- Only applied to expressions with integer type of rank lower than `(unsigned)int`
- Converted to `int`, if representable by that
- Otherwise, converted to `unsigned int`
- Applied for:
  - usual arithmetic conversions
  - default argument promotions
  - operand of unary +, - and ~ operators
  - both operands of the shift operators

Default argument promotions

Applied to arguments if no type for the corresponding parameter is specified

1. Apply integer promotions
2. Convert `float`s to `double`s

```
void f(); // Arbitrary number of parameters

void g(void); // No parameters

int printf(char const *format, ...); // Variadic
```

Default argument promotions

Applied to arguments if no type for the corresponding parameter is specified

1. Apply integer promotions
2. Convert `float`s to `double`s

```
void f(); // Arbitrary number of parameters

void g(void); // No parameters

int printf(char const *format, ...); // Variadic
```

Default argument promotions

Applied to arguments if no type for the corresponding parameter is specified

1. Apply integer promotions
2. Convert `float`s to `double`s

```c
void f(); // Arbitrary number of parameters

void g(void); // No parameters

int printf(char const *format, ...); // Variadic
```

Usual arithmetic conversions

Applied for certain operations:

- multiplicative (`*`, `/`, `%`)
- additive (`+`, `-`)
- relational (`<`, `>`, `<=`, `>=`)
- equality (`==`, `!=`)
- bitwise (`&`, `|`, `^`)
- conditional (`a ? b : c`, only to the second and third operand)

Usual arithmetic conversions

1. If one operand’s type is **`long double`**, the other is converted to **`long double`**
2. Otherwise, if one operand’s type is **`double`**, the other is converted to **`double`**
3. Otherwise, if one operand’s type is **`float`**, the other is converted to **`float`**

Usual arithmetic conversions

4. Otherwise, the integer promotions are performed on both operands. If the types are equal after this the conversion is finished
5. Otherwise, if both operands have the same signedness, the operand with the type of lesser integer conversion rank is converted to the type of the other operand
6. Otherwise, if the operand with unsigned integer type has a type with greater rank than the signed operand, the signed operand is converted to the type of the unsigned operand

Usual arithmetic conversions

7. Otherwise, if the type of the operand with signed integer type can represent all values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type
8. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type

Usual arithmetic conversions - Example

```c
unsigned int a = 1;
signed int b = -1, c = a + b;
if (a > b) printf(”True\n”);
```

- a and b have the same rank
- For both + and >, b is converted to `unsigned int`
- Effect: `a > b` is false
- $a + b_{\text{signed}} \equiv a + b_{\text{unsigned}} \ (\mathrm{mod}\ (\text{UINT\_MAX} + 1))$, hence `a + b` is 0

Usual arithmetic conversions - Example

```
unsigned int a = 1;
signed int b = -1, c = a + b;
if (a > b) printf(”True\n”);
```

- a and b have the same rank
- For both + and >, b is converted to **unsigned int**
- Effect: a > b is false
- $a + b_{\text{signed}} \equiv a + b_{\text{unsigned}} \pmod{\texttt{UINT\_MAX} + 1}$, hence a + b is 0

Usual arithmetic conversions - Example

```c
unsigned int a = 1;
signed int b = -1, c = a + b;
if (a > b) printf(”True\n”);
```

- a and b have the same rank
- For both + and >, b is converted to `unsigned int`
- Effect: `a > b` is false
- $a + b_{\text{signed}} \equiv a + b_{\text{unsigned}} \pmod{\mathtt{UINT\_MAX} + 1}$, hence `a + b` is 0

Usual arithmetic conversions - Example

```c
unsigned int a = 1;
signed int b = -1, c = a + b;
if (a > b) printf(”True\n”);
```

- a and b have the same rank
- For both + and >, b is converted to `unsigned int`
- Effect: `a > b` is false
- $a + b_{\text{signed}} \equiv a + b_{\text{unsigned}} \pmod{(\texttt{UINT\_MAX} + 1)}$, hence `a + b` is 0

1. Undefined behaviour
2. Strict aliasing
3. Arrays
4. Conversions
5. **Fun with C99 (and above)**

How many possible results does this function have?

```c
int f(signed int *i1, unsigned int *i2, float *f, char *c) {
  *i1 = 42;
  *i2 = 43;
  *f = 13.;
  *c = 1;
  return *i1 + *i2 + *f + *c;
}
```

19

1

2^{104}

214

How many possible results does this function have?

```c
int f(signed int *i1, unsigned int *i2, float *f, char *c) {
  *i1 = 42;
  *i2 = 43;
  *f = 13.;
  *c = 1;
  return *i1 + *i2 + *f + *c;
}
```

[✓ 19]

1

2^{104}

214

```c
int f(signed int *i1,
      unsigned int *i2,
      float *f,
      char *c) {
  *i1 = 42;
  *i2 = 43;
  *f = 13.;
  *c = 1;
  return *i1 + *i2 + *f + *c;
}
```

```asm
.LCPI0_0:
  .long 1065353216 # float 1
f:
  movl $42, (%rdi)
  movl $43, (%rsi)
  movl $1095761920, (%rdx)
  movb $1, (%rcx)
  movl (%rsi), %eax
  addl (%rdi), %eax
  cvtsi2ssq %rax, %xmm0
  addss (%rdx), %xmm0
  addss .LCPI0_0(%rip), %xmm0
  cvttss2si %xmm0, %eax
  retq
```

restrict

- C99 added the `restrict` qualifier
- can only be applied to pointer types
- the pointee may only be accessed via an expression based on the pointer
- restricts aliasing

restrict

- C99 added the **`restrict`** qualifier
- can only be applied to pointer types
- the pointee may only be accessed via an expression based on the pointer
- restricts aliasing

restrict

- C99 added the `restrict` qualifier
- can only be applied to pointer types
- the pointee may only be accessed via an expression based on the pointer
- restricts aliasing

restrict

- C99 added the `restrict` qualifier
- can only be applied to pointer types
- the pointee may only be accessed via an expression based on the pointer
- restricts aliasing

```c
int f(signed int *restrict i1,
      unsigned int *restrict i2,
      float *restrict f,
      char *restrict c) {
  *i1 = 42;
  *i2 = 43;
  *f = 13.;
  *c = 1;
  return *i1 + *i2 + *f + *c;
}
```

```asm
f:
  movl $42, (%rdi)
  movl $43, (%rsi)
  movl $1095761920, (%rdx)
  movb $1, (%rcx)
  movl $99, %eax
  retq
```

Compound literals

- Anonymous objects in C
- Look like casting an initializer: `int *A = (int[3]){42, 3, 5}`
- Are L-values: `(char){’a’} = ’b’`

```
GPIO_Init(GPIOD, &(GPIO_InitTypeDef){
    .GPIO_Pin = GPIO_Pin_4,
    .GPIO_Mode = GPIO_Mode_OUT,
    .GPIO_OType = GPIO_OType_PP,
    .GPIO_PuPd = GPIO_PuPd_NOPULL,
    .GPIO_Speed = GPIO_Speed_50MHz
});
```

Compound literals

- Anonymous objects in C
- Look like casting an initializer: `int *A = (int[3]){42, 3, 5}`
- Are L-values: `(char){’a’} = ’b’`

```
GPIO_Init(GPIOD, &(GPIO_InitTypeDef){
  .GPIO_Pin = GPIO_Pin_4,
  .GPIO_Mode = GPIO_Mode_OUT,
  .GPIO_OType = GPIO_OType_PP,
  .GPIO_PuPd = GPIO_PuPd_NOPULL,
  .GPIO_Speed = GPIO_Speed_50MHz
});
```

Compound literals

- Anonymous objects in C
- Look like casting an initializer: `int *A = (int[3]){42, 3, 5}`
- Are L-values: `(char){'a'} = 'b'`

```
GPIO_Init(GPIOD, &(GPIO_InitTypeDef){
  .GPIO_Pin = GPIO_Pin_4,
  .GPIO_Mode = GPIO_Mode_OUT,
  .GPIO_OType = GPIO_OType_PP,
  .GPIO_PuPd = GPIO_PuPd_NOPULL,
  .GPIO_Speed = GPIO_Speed_50MHz
});
```

Compound literals

- Anonymous objects in C
- Look like casting an initializer: `int *A = (int[3]){42, 3, 5}`
- Are L-values: `(char){’a’} = ’b’`

```
GPIO_Init(GPIOD, &(GPIO_InitTypeDef){
    .GPIO_Pin = GPIO_Pin_4,
    .GPIO_Mode = GPIO_Mode_OUT,
    .GPIO_OType = GPIO_OType_PP,
    .GPIO_PuPd = GPIO_PuPd_NOPULL,
    .GPIO_Speed = GPIO_Speed_50MHz
});
```

Booleans

- A real boolean type called `_Bool` exists
- Easy usage using `stdbool.h`
- Typedef called `bool`
- Constants `true` and `false`

Booleans

- A real boolean type called `_Bool` exists
- Easy usage using `stdbool.h`
- Typedef called `bool`
- Constants `true` and `false`

Booleans

- A real boolean type called `_Bool` exists
- Easy usage using `stdbool.h`
- Typedef called `bool`
- Constants `true` and `false`

Booleans

- A real boolean type called **`_Bool`** exists
- Easy usage using `stdbool.h`
- Typedef called `bool`
- Constants `true` and `false`

Thank you for your attention.
Any questions?

http://babelmonkeys.de/~florob/talks/AC-2015-03-25-undefC.pdf