

# Intermezzo Exam Date

Doodle to find exam date (participation required)

<https://doodle.com/poll/5qgu4tbv5ig6aig8>

Deadline: 8th December 2017

The screenshot shows a Doodle poll interface. At the top, there's a navigation bar with 'Doodle', 'Preise', 'Hilfe', 'Deutsch', a user profile for 'Marcus Pinnecke', and a button to 'Doodle erstellen'. Below the navigation is a red button with the text '★ Endgültige Option wählen'. To the right of this button are links for 'Einladen' and 'Mehr'. The main title of the poll is 'Exam Date "ARCADE"' by Marcus Pinnecke, posted 4 minutes ago. Below the title, there are three informational icons: a location pin labeled 'TBA', a list icon with the text 'Exam date for per-student oral exam and per-project-group oral exam/presentation', and a clock icon with the text 'Alle Zeiten angezeigt in Europe/Berlin'. At the bottom, there are two tabs: 'Tabelle' (selected) and 'Kalender'. The table view shows a grid of dates from February 12 to 19, 2018, with specific times listed for each slot.

	Feb 12 MO	Feb 12 MO	Feb 13 DI	Feb 13 DI	Feb 15 DO	Feb 15 DO	Feb 19 MO	Fe
08:00 – 11:00	12:00 – 14:00	08:00 – 11:00						

1 Scopes

Lookup & Namespaces

2

Storage Classes

5

Linkage

3

Storage Duration

4

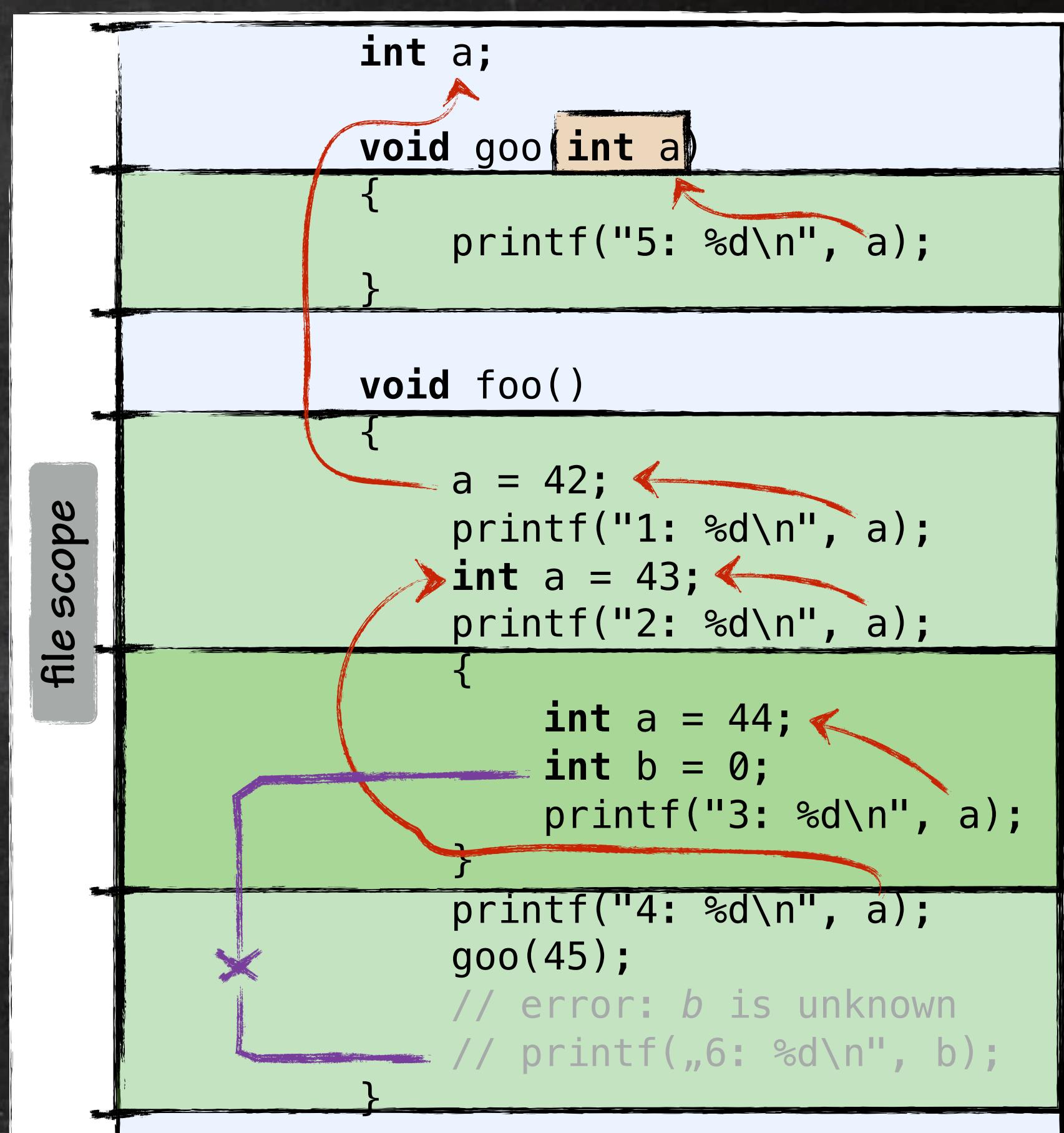
# Scopes

Scope is a particular subset of the source code in which an identifier is visible.

## Scopes in C:

- File scope
- Block scope
- Function scope [not in Figure]
- Function prototype scope

Nested scopes: if multiple entities named by the same identifier are in scope and if these entities belong to the same namespace (see next slide), declaration of inner scope hides declaration of outer scope.



# Lookup & Namespaces

Note the difference to the term namespace in other languages like C++:  
namespaces are not user-defined in C!

C supports identical identifiers to name different entities in the same scope if these entities are in different namespaces.

A namespace (in C) is a particular language-defined category:

- Label namespace for labels for jumping, like goto: `foo:`
- Tag namespace for naming enum, structs or unions: `enum foo /* ... */;`
- Per-struct/per-union member namespace: `struct s1 {int foo;}; struct s2 {int foo;};`
- Others
  - Function names: `void foo (void);`
  - Object names: `int foo;`
  - Typedef names: `typedef int foo;`
  - Enumeration constant names: `enum e1 {foo};`

# Linkage

Referencing identifier from other scopes is called **linkage**.

## no linkage

Referencing only in scope the identifier is in.

applies to function argument list

```
void goo( int a ) { /* ... */ }
```

applies to regular block-scope variables

```
{
    int b = 0;
}
```

// b is unknown here

## internal linkage

Referencing from all scopes in current translation unit (keyword **static**).

**foo.c**

```
static int private_foo;
static void private_func(void);
/* ... */
must be declared in file scope
```

**goo.c**

```
/* private_foo and private_func
are unknown here */
```

## external linkage

Referencing from all scopes from all translation units.

**foo.c**

```
int public_foo;
int public_foo_2;
void public_func(void);
```

**foo.h**

```
extern int public_foo_2;
```

**goo.c**

```
/* public_foo, public_foo_2,
and public_func are known
here */
```

# Storage Duration

Storage duration determines the lifetime of objects.

Types: **automatic**, **static**, **thread**, and **allocated**.

controlled by the programmer

allocated

```
void *foo = malloc(/*...*/)
```

foo's value lifetime starts here

storage allocation here

```
free(foo);
```

foo's value lifetime ends here

storage deallocation here

controlled by the programmer

automatic

compound statement  
„block“

```
/* ... */  
{  
    storage allocation here  
    ↑ foo's lifetime starts here  
    int foo;  
    ↓ foo's lifetime ends here  
}  
    storage deallocation here  
/* ... */
```

static

```
storage allocation here  
↑ foo's lifetime starts here  
/* ... */  
static int foo;  
/* ... */  
storage deallocation here  
↓ foo's lifetime ends here
```

main is invoked

main returns

thread-local

```
start  
↑ foo's lifetime starts here  
storage allocation here  
/* ... */  
_Thread_local int foo;  
/* ... */  
end  
↓ foo's lifetime ends here  
storage deallocation here
```

- Object lifetime bound to block

- Objects initialized once before main is entered
- Object lifetime bound to program lifetime

- Each thread has own distinct object
- Object lifetime bound to thread lifetime

# Storage Class

**Storage class** specifies **storage duration** and **linkage** of functions and objects in C.  
Specified by keywords **auto** , **extern** , **static** and **register** .

Storage Class	Place	Initial Value	Duration	Linkage	Declaration
<b>auto</b>	stack memory	garbage	automatic	no linkage	only on objects, only at block scope or function parameter list
<b>extern</b>	data segment	zero	static	external	functions & objects, only at file and block scope
<b>static</b>	data segment	zero	static	internal	functions: at file scope variables: file and block scope
<b>register</b>	stack memory or CPU register	garbage	automatic	no linkage	only on objects, only at block scope or function parameter list

# Keywords

already covered

soon...

not consider in this lecture

auto

else

long

struct

\_Alignof

break

enum

register

switch

\_Atomic

case

extern

restrict

typedef

\_Bool

char

float

return

union

\_Complex

const

for

short

unsigned

\_Generic

continue

goto

signed

void

\_Imaginary

default

if

sizeof

volatile

\_Noreturn

do

inline

static

while

\_Static\_assert

double

int

asm

\_Alignas

\_Thread\_local

## Part III

# Expressions

Expressions

Value Categories

Assignment Operators

Increment/Decrement Operators

Arithmetic Operators

Comparison Operators

Logical Operators

Bitwise Operators

Member Access Operators

Other Operators

Precedences

# Expressions

An **expression** is a **combination** of at least one **value, constant, variable, operator, and function** to perform a **computation**, i.e., to **evaluate another value**.

Order of evaluation (evaluation of operands, subexpressions,...) is **determined by**

- **precedence rules**
- **association rules.**

```
(x + 3 * f(x));
```

example

# Value Categories (1)

Ivalue

One property of expressions in C is **value category classes**: lvalue, rvalue, and function designator

- **Ivalue expressions** („left value“)
  - any expression **not returning void** i.e., a particular value is returned:
    - identifiers, e.g., `i`
    - string literals, e.g., `"Hello World"`
    - compound literals, e.g., `(const char []){"abc"}`
    - parenting lvalue expressions, e.g., `(i + 5)`
    - result of member access operator if left-side operand is lvalue, e.g., `my_struct.x`
    - result of member access operator by pointer, e.g., `my_struct_ptr->x`
    - result of dereferencing a memory address, e.g., `*my_struct_ptr`
    - result of array element getter operator, e.g., `my_array[23]`
  - the following cases, lvalue expressions may be used as replacement for `x`
    - the operand for taking the memory address, i.e., `&x`
    - the operand for increment/decrement operators, i.e., `i++`
    - left-side operand for member access operator, i.e., `x.y`
    - left-side operand for assignment, i.e., `x = x + y`
    - left-side operand for compound assignment, i.e., `x += y`
  - In most cases, lvalue expression lead directly to a value memory load

## Value Categories (2)

Ivalue

One property of expression in C is **value category classes**: lvalue, rvalue, and function designator

- **Ivalue expressions** („left value“)
  - Ivalue expressions are **mutable** unless
    - they are from type **array** and marked as **immutable** (via **const** keyword)
    - they are of an **incomplete type** (see later)
    - they are a **struct/union** type having (recursively) **one immutable member** (via **const**)

## Value Categories (3)

rvalue

One property of expression in C is **value category classes**: lvalue, rvalue, and function designator

- **rvalue expressions** („right value“)
  - expressions that evaluate to values that have no object identity or storage location:
    - constants (integer, character, constants), e.g., 23L
    - all operators that do not return lvalues, examples (not complete):
      - case expression, e.g., (float)
      - arithmetic, comparison, logical and bitwise operators, e.g., +
      - increment/decrement operators, e.g., ++
      - assignment & compound assignment operators, e.g., +=
      - ...
  - address of a rvalue expressions cannot be taken

## Value Categories (4)

function  
designator

One property of expression in C is **value category classes**: lvalue, rvalue, and function designator

- **function designator** expressions (function name)
  - expression of type function
  - function designators convert into pointer to function

```
void foo() { }
main() { printf("%p", foo); }
```

Output example  
0x105664e70

...  
%p in printf is  
the format to  
print memory  
addresses

# Arithmetic Operators

- $+y$
- $-y$
- $x + y$
- $x - y$
- $x * y$
- $x / y$
- $x \% y$

**unary plus:** the value of  $y$

**unary minus:** the negative value of  $y$

**addition:** the addition of  $x$  and  $y$

**subtraction:** the subtraction of  $y$  from  $x$

**product:** the product of  $x$  and  $y$

**division:** the division of  $x$  by  $y$

**modulo:** the remainder of division of  $x$  by  $y$

# Increment/Decrement Operators

y must be a mutable lvalue expression resulting in number or pointer type	• <b>y++</b>	postfix increment: y gets its value plus 1	→ evaluates to value of y
	• <b>y--</b>	postfix decrement: y gets its value minus 1	
	• <b>++y</b>	prefix increment: y gets its value plus 1	→ evaluates to value of y+=1 resp. y-=1
	• <b>--y</b>	prefix decrement: y gets its value minus 1	

value of **y** is  
incremented/  
decremented  
by 1

## Example

```
int a = 42;
int b = a++; // b == 42 and a == 43

int a = 42;
int b = ++a; // b == 43 and a == 43
```

# Comparison Operators

Comparison („relational“) operators result in boolean value:

1 (true) if condition holds  
0 (false) otherwise

- $x == y$  equality: is  $x$  equal to  $y$ ?
- $x != y$  inequality: is  $x$  not equal to  $y$ ?
- $x < y$  less than: is  $x$  less than  $y$ ?
- $x > y$  greater than: is  $x$  greater than  $y$ ?
- $x <= y$  less than or equal to: is  $x$  less than or equal to  $y$ ?
- $x >= y$  greater than or equal to: is  $x$  greater or equal to  $y$ ?

# Logical Operators

Implementation of standard boolean algebra operations

- `!y` logical NOT: logical negation of y
- `x && y` logical AND: logical AND of x and y
- `x || y` logical OR: logical OR of x and y

# Bitwise Arithmetic Operators

## Implementation of standard bitwise arithmetic operators

- $\sim y$       bitwise NOT: bitwise NOT of y
- $x \& y$       bitwise AND: bitwise AND of x and y
- $x | y$       bitwise OR: bitwise OR of x and y
- $x ^ y$       bitwise XOR: bitwise XOR of x and y
- $x << y$       bitwise left shift : x left shifted by y
- $x >> y$       bitwise right shift: x right shifted by y

# Assignment Operators

## compound assignment

- `y = x` basic assignment: *y gets the value of x*  
*y is mutable lvalue expression and x is implicitly convertible to or compatible with type of y*
- `y += x` addition assignment: *y gets its own value plus x*
- `y -= x` subtraction assignment: *y gets its own value minus x*
- `y *= x` multiplication assignment: *y gets its own value multiplied by x*
- `y /= x` division assignment: *y gets its own value divided by x*
- `y %= x` modulo assignment: *y gets its own value modulo x*
- `y &= x` bitwise AND assignment: *y gets the value of bitwise AND of its own value and x*
- `y |= x` bitwise OR assignment: *y gets the value of bitwise OR of its own value and x*
- `y ^= x` bitwise XOR assignment: *y gets the value of bitwise XOR of its own value and x*
- `y <<= x` bitwise left shift assignment: *y gets the value y left shifted by x*
- `y >>= x` bitwise right shift assignment: *y gets the value y right shifted by x*

# Member Access Operators

Member access operators give access to members of their operands (arrays, pointers,...)

- `x [y]` array subscript: get the y-th element in array x
- `*y` pointer dereference: dereference pointer to y in order to get object/function at address stored in y
- `&y` address of: get (memory) address of object/function and create pointer with that address
- `x . y` member access: get the member called y in the struct/union x
- `x->y` member access by pointer: get member called y in struct/union which is pointed by x

more details here,  
when we talk about  
arrays and pointers

# Other Operators

- `f( ... )`
- `x, y`
- `(x)y`
- `x?y:z`
- `sizeof(y)`
- `_Alignof(y)`

**function call:** call function *f* with arguments given by ...

**comma:** evaluate expression *x*, throw away its result, evaluate expression *y*, return its type and value from evaluation of *y* `int y; foo((y=3, y+2));` ≡ `int y = 3; foo((y+2));`

**type cast:** perform explicit type conversion for *y* to type *x*

**conditional operator:** if expression *x* evaluates to non-zero, evaluate *y* otherwise *z*

**sizeof operator:** get size in bytes of *y*

**alignof operator:** get alignment requirements of *y*

taken from: [http://en.cppreference.com/w/c/language/operator\\_other](http://en.cppreference.com/w/c/language/operator_other)

# Precedences

A table for operator precedences can be found here:

[http://en.cppreference.com/w/c/language/operator\\_precedence](http://en.cppreference.com/w/c/language/operator_precedence)