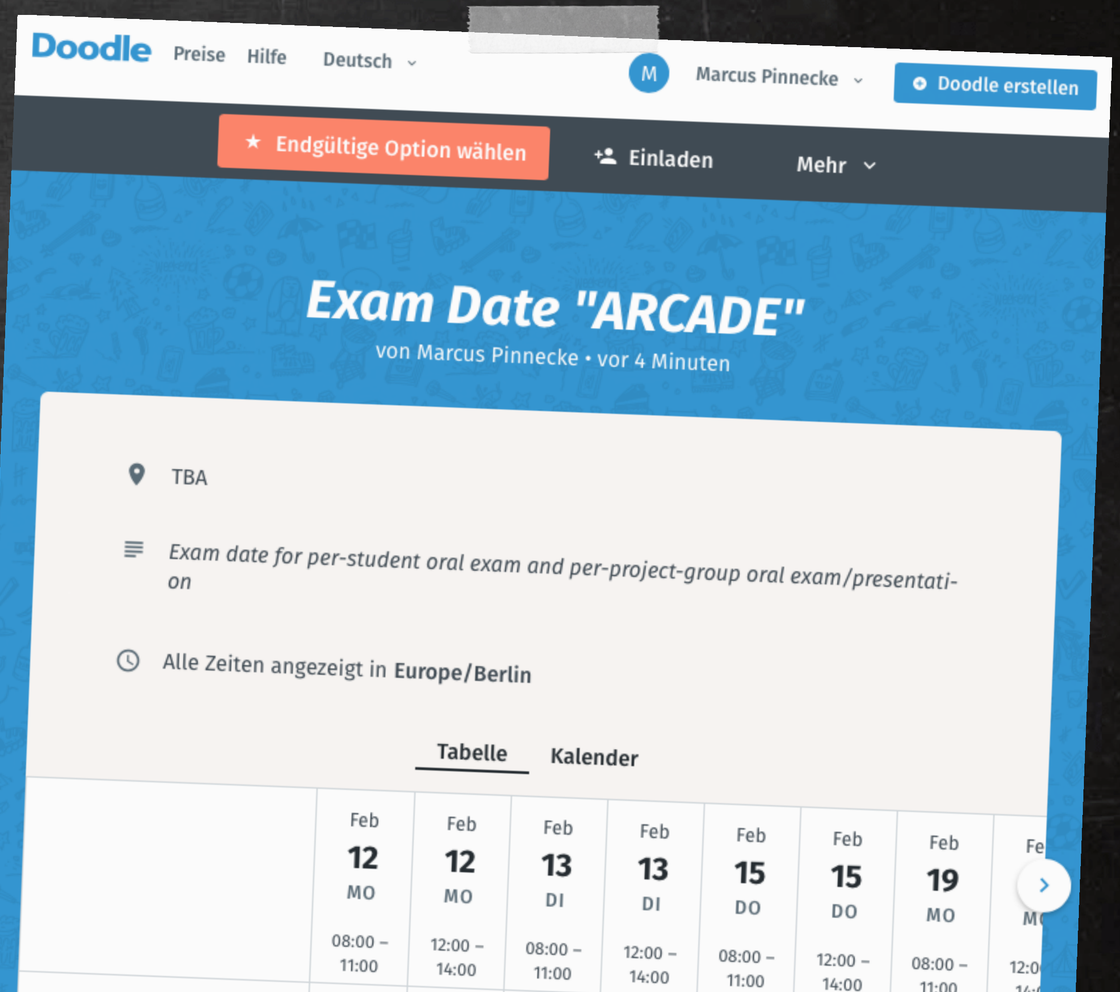
Intermezzo

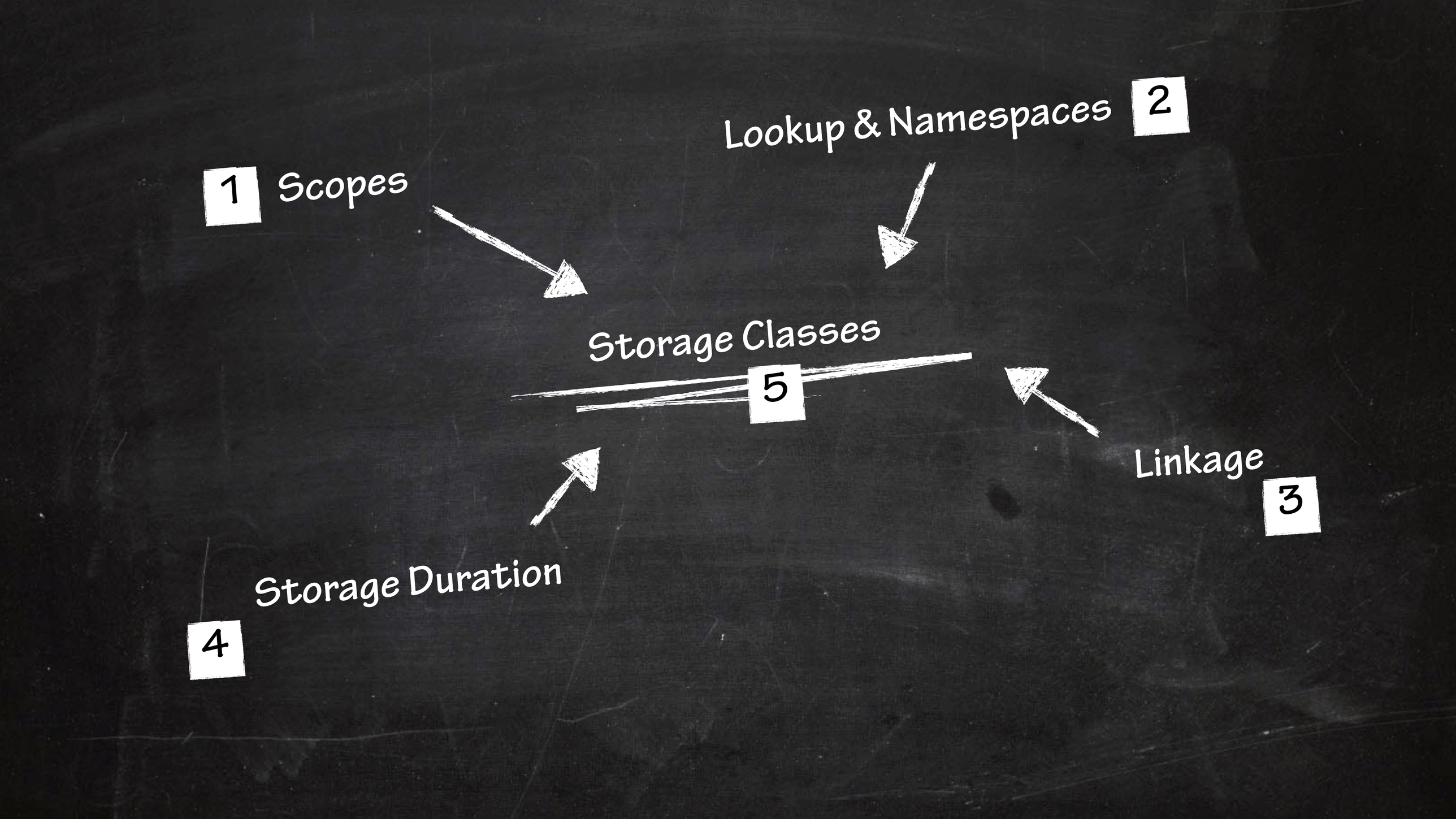
Exam Date

Doodle to find exam date (participation required)

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

Deadline: 8th December 2017




1 Scopes
Lookup & Namespaces 2
Storage Classes
5
Linkage
3
Storage Duration
4

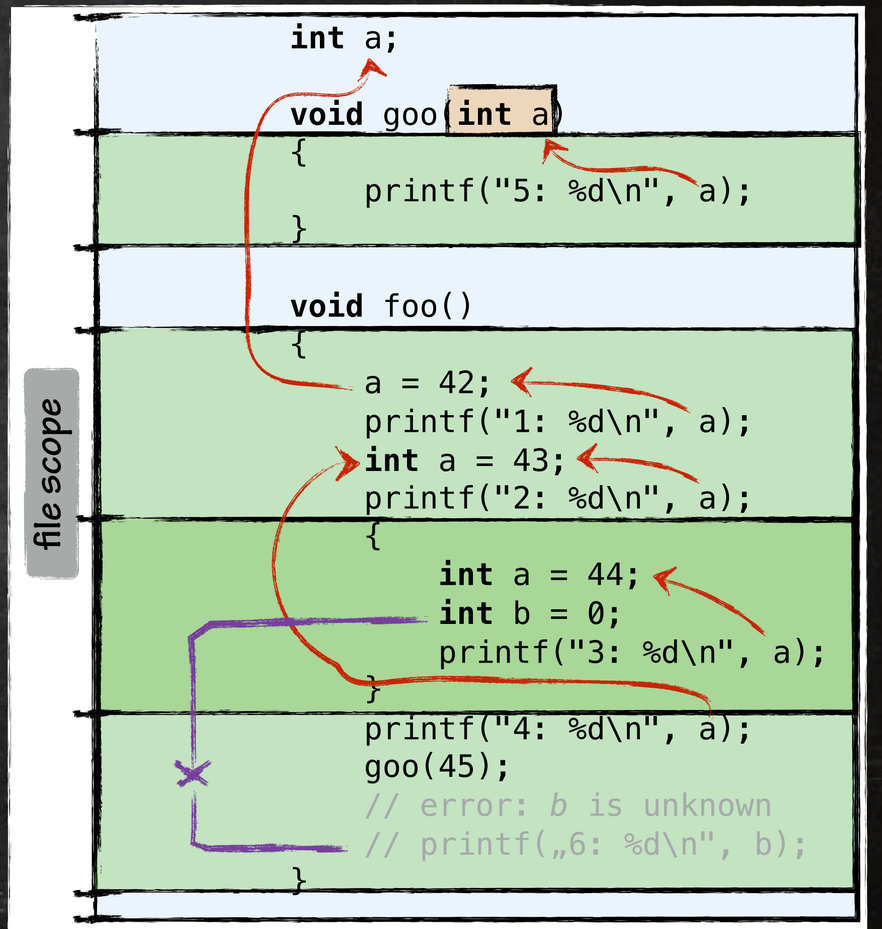
Scopes

Output is
1: 42
2: 43
3: 44
4: 43
5: 45

Scope is 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 **same namespace** (see next slide), **declaration** of **inner** scope **hides** declaration of **outer** scope.

based on: http://en.cppreference.com/w/c/language/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 an different entities in the same scope if these entities are in different namespaces.

A namespace (in C) in 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: `typdef int foo;`
  - Enumeration constant names: `enum e1 {foo};`

taken from: http://en.cppreference.com/w/c/language/name_space

based on: http://en.cppreference.com/w/c/language/storage_duratio

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;
```

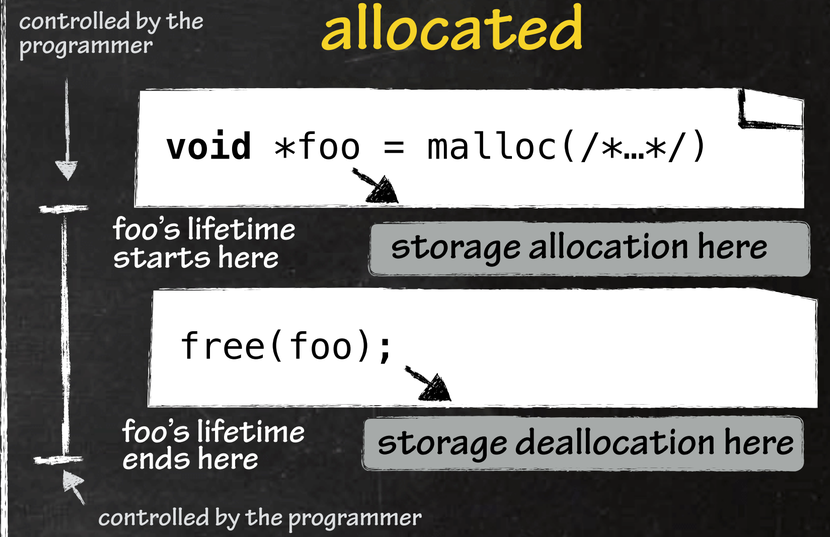
goo.c

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

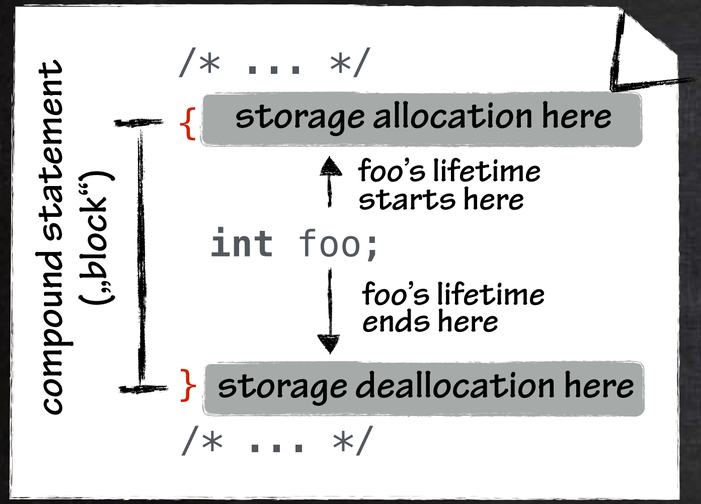
Storage Duration

Storage duration determines the lifetime of objects.
Types: automatic, static, thread, and allocated.

allocated

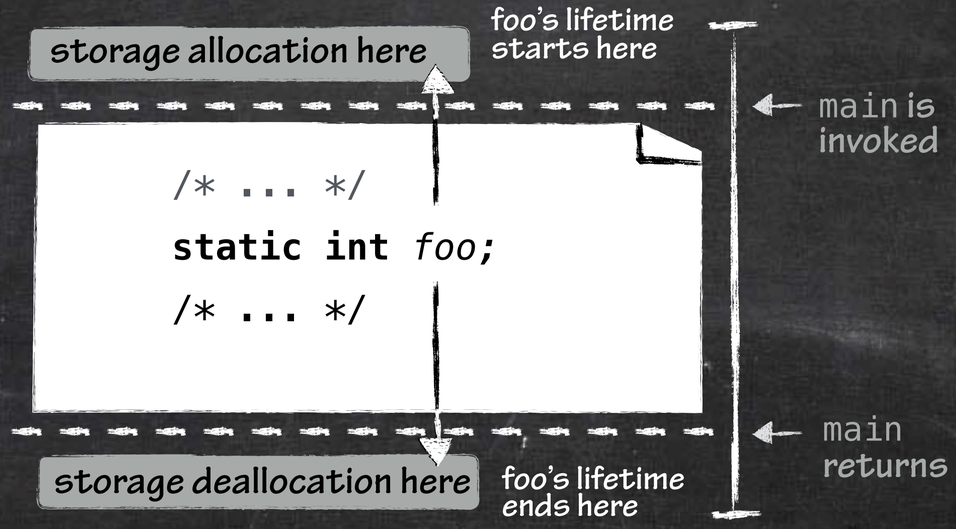


automatic



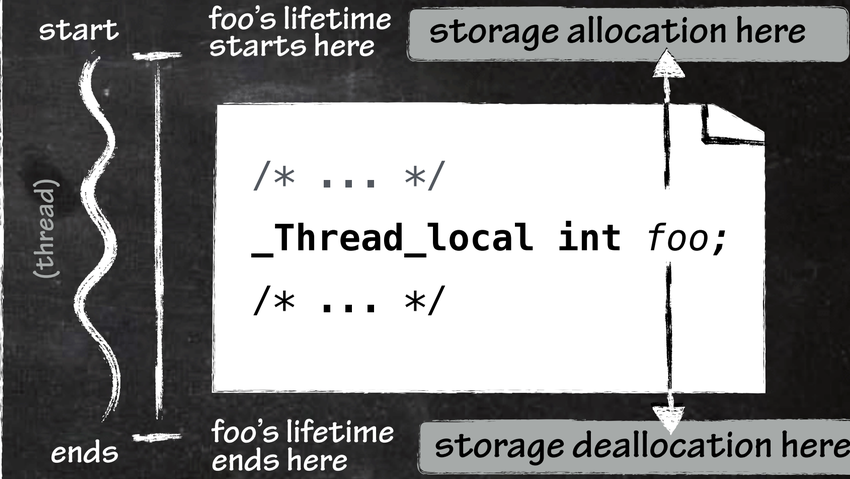
- Object lifetime bound to block

static



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

thread-local



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

based on: http://en.cppreference.com/w/c/language/storage_duration

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
`auto`	stack memory	garbage	automatic	no linkage	only on objects, only at block scope or function parameter list
`extern`	data segment	zero	static	external	functions & objects, only at file and block scope
`static`	data segment	zero	static	internal	functions: at file scope variables: file and block scope
`register`	stack memory *or* CPU register	garbage	automatic	no linkage	only on objects, only at block scope or function parameter list

Keywords

already covered

not consider in this lecture

soon...

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 *values*, *constants*, *variables*, *operators*, and *functions* 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)

lvalue

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

- lvalue 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

taken from: http://en.cppreference.com/w/c/language/value_category

Value Categories (2)

lvalue

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

- lvalue expressions („left value“)
  - lvalue 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 an struct/union type having (recursively) one *immutable* member (via `const`)

based on: http://en.cppreference.com/w/c/language/value_category

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

taken from: http://en.cppreference.com/w/c/language/value_category

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

taken from: http://en.cppreference.com/w/c/language/value_category

Assignment Operators

- `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

compound assignment

- `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

taken from: http://en.cppreference.com/w/c/language/operator_assignment

Increment/Decrement Operators

- `y++` **postfix increment** y gets its value **plus 1**
- `y--` **postfix decrement** y gets its value **minus 1**

→ results in value of `y`

y must be a mutable lvalue expression resulting in number or pointer type

- `++y` **prefix increment** y gets its value **plus 1**
- `--y` **prefix decrement** y gets its value **minus 1**

→ results in value of `y+=1` resp. `y-=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
```

based on: http://en.cppreference.com/w/c/language/operator_assignment

Arithmetic Operators

- `+y` unary plus: the value of y
- `-y` unary minus: the negative value of y
- `x + y` addition: the addition of x and y
- `x - y` subtraction: the subtraction of y from x
- `x * y` product: the product of x and y
- `x / y` division: the division of x by y
- `x % y` modulo: the remainder of division of x by y

based on: http://en.cppreference.com/w/c/language/operator_assignment

Comparison Operators

Comparison („relational“) operators result in boolean value:
1 (`true`) if conditions 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?

based on: http://en.cppreference.com/w/c/language/operator_assignment

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

based on: http://en.cppreference.com/w/c/language/operator_assignment

Bitwise Arithmetic Operators

Implementation of standard bitwise arithmetic operators

- `~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

based on: http://en.cppreference.com/w/c/language/operator_assignment

Member Access Operators

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

more details here, when we talk about arrays and 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

based on: http://en.cppreference.com/w/c/language/operator_assignment

Other Operators

- `f(…)` — function call: call function *f* with arguments given by …
- `x, y` — 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));`
- `(x)y` — type cast: perform explicit type conversion for *y* to type *x*
- `x?y:z` — conditional operator: if expression *x* evaluates to non-zero, evaluate *y* otherwise *z*
- `sizeof(y)` — sizeof operator: get size in bytes of *y*
- `_Alignof(y)` — alignof operator: get alignment requirements of *y*

taken from: 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