Part II

Basic Concepts

Comments | Types | Type Conversions | Objects, Padding, Packing | Naming, Scoping & Lifetimes

Lookup and Namespaces | Storage Classes | Keywords

Comments (1)

Description and documentation that are ignored by the compiler.
C supports two types

1 **Multi-line comments**

```
/* this is a
   multi-line comment */
```

- used to comment larger blocks or as single line comment („C-style“ comment)
- comment starts with first occurrence of `/*` until `*/` is found
- often used for code documentation (e.g., via doxygen)

```
/**
 * (docu)
 */
```

JavaDoc style

```
/*!
 * (docu)
 */
```

Qt style

...

optional

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

Comments (2)

Description and documentation that are ignored by the compiler.
C supports two types

2 **Single-line comments**

```
// this is a single-line comment
```

- used to comment a line of code („C++ -style“ comment)
- comment starts with first occurrence of `//` until end of line
- also used for code documentation (e.g., via doxygen)

```
///
///(docu)
///
```

JavaDoc style

```
//!
//! (docu)
//!
```

Qt style

...

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

Comments (3)

Examples:

```
// Copyright (C) 2017 (Name)
//
// This program is free software: you can redistrib
// GNU General Public License as published by the F
// (at your option) any later user_port.
//
// This program is distributed in the hope that it
// warranty of MERCHANTABILITY or FITNESS FOR A PAR
// See the GNU General Public License for more deta
//
// You should have received a copy of the GNU Gener
// If not, see <http://www.gnu.org/licenses/>.
```

per-file licensing information

```
// ------------------------------------------------
// I N C L U D E S
// ------------------------------------------------

/* ... */

// ------------------------------------------------
// D A T A   T Y P E S
// ------------------------------------------------

/* ... */

// ------------------------------------------------
// I N T E R F A C E   F U N C T I O N S
```

file structuring

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

Comments (4)

Examples:

```
typedef struct gs_tuplet_t {
    struct gs_frag_t *fragment; /*<! fragment in which this tuplet exists */
    gs_tuplet_id_t tuplet_id; /*<! number of this tuplet inside the fragment */
    void *attr_base; /*<! pointer in fragment where first attribute of this tuplet is located */

    /* operations */
    bool (*_next)(struct gs_tuplet_t *self); /* seeks to the next tuplet inside this fragment */
    void (*_open)(struct gs_tuplet_field_t *dst, struct gs_tuplet_t *self); /*<! access the attr_value_ptr data of this
    void (*_update)(struct gs_tuplet_t *self, const void *data); /*<! updates all fields of this tuplet and moves to nex
    void (*_set_null)(struct gs_tuplet_t *self); /*<! updates all fields of this tuplet to NULL, and moves to next */
    void (*_delete)(struct gs_tuplet_t *self); /* request to delete this tuplet from fragment */
    bool (*_is_null)(struct gs_tuplet_t *self); /*<! checks whether this tuplet is NULL entirely */
} gs_tuplet_t;
```

explanation of members and their intention

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

Comments (5)

```
static inline gs_tuplet_id_t gs_global_to_local(gs_grid_t *
{
    // TODO: apply a bunch of strategy alternatives here.
    …
}
```

annotate ToDo's, Fixes,… (often backed by IDE)

```
// Determine the end of the interval list. If cursor reach
// Clearly, if this case happens, there is an interval gs_
// a valid coverage of the given 'tuple_id' in this grid
const gs_tuple_id_interval_t *end = gs_vec_end(grid->tuple

if (!INTERVAL_CONTAINS(cursor, tuple_id)) {
   switch (type) {
       case AT_SEQUENTIAL:
           // seek to interval that contains the tuple, mo
           // current one
           for (cursor++; cursor < end && !INTERVAL_CONTAI
```

give information on intended functionality, usage, side effects,… in implementation for non-trivial passages

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

Comments (6)

... and sometimes for jokes

```
long long ago; /* in a galaxy far far away */
```

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

Types

A type is used to state how a particular bunch of binary data should be interpreted.

```
01 A3 12 F1 04 55 12 AF
```

Types in C11

- void
- basic types
- enumerated types
- composite/derived types

Types

A type is used to state how a particular bunch of binary data should be interpreted.

```
01 A3 12 F1 04 55 12 AF
```

basic types

basic types - actual value ranges are implemented-dependent!

- `char` smallest addressable unit to present a character
  - typically 1 byte but has actually `CHAR_BIT` bits (see std-lib)
- `_Bool` a boolean type to store logical values (1 = true and 0 = false)
- `int` a (signed) integer value with the natural size on the host machine
  - at least 2 byte & contains at least values in [-32'767, 32'767]
- `float` a single-precision floating point number
  - often IEEE 754 single-precision binary floating-point format is used
- `double` a double-precision floating point number
  - IEEE 754 double-precision binary floating-point format

C language specification only defines lower bounds for bytes used for basic types.

Types

A type is used to state how a particular bunch of binary data should be interpreted.

```
01 A3 12 F1 04 55 12 AF
```

basic types

basic types - actual value ranges are implemented-dependent!

C language specification only defines lower bounds for bytes used for basic types.

+ implementation-defined types (e.g., `__uint128` with 16 Byte)
+ complex types (`_Complex`) and imaginary types (`_Imaginary`)

Types

A type is used to state how a particular bunch of binary data should be interpreted.

```
01 A3 12 F1 04 55 12 AF
```

basic types

basic types - actual value ranges are implemented-dependent!

Qualifiers (`short` and `long`) applied to basic number types

- `short int` at least 2 bytes
- `int` at least as many bytes as short (but often 4 bytes)
- `long int` at least as many bytes as int (but often 8 bytes)
- `long double` states extended-precision floating point (implementation-defined)

Overview

basic types

1. `char`
2. `signed char` at least [-127, +127]
3. `unsigned char` at least [0, +255]

>= 16bit

4. `short` = `short int` = `signed short` = `signed short int` at least [-32'767, +32'767]
5. `unsigned short` = `unsigned short int` at least [0, +65'535]
6. `int` = `signed` = `signed int` at least [-32'767, +32'767]
7. `unsigned` = `unsigned int` at least [0, +65'535]

>= 32bit

8. `long` = `long int` = `signed long` = `signed long int` at least [-2'147'483'647, +2'147'483'647]
9. `unsigned long` = `unsigned long int` at least [0, +4'294'967'295]

>= 64bit

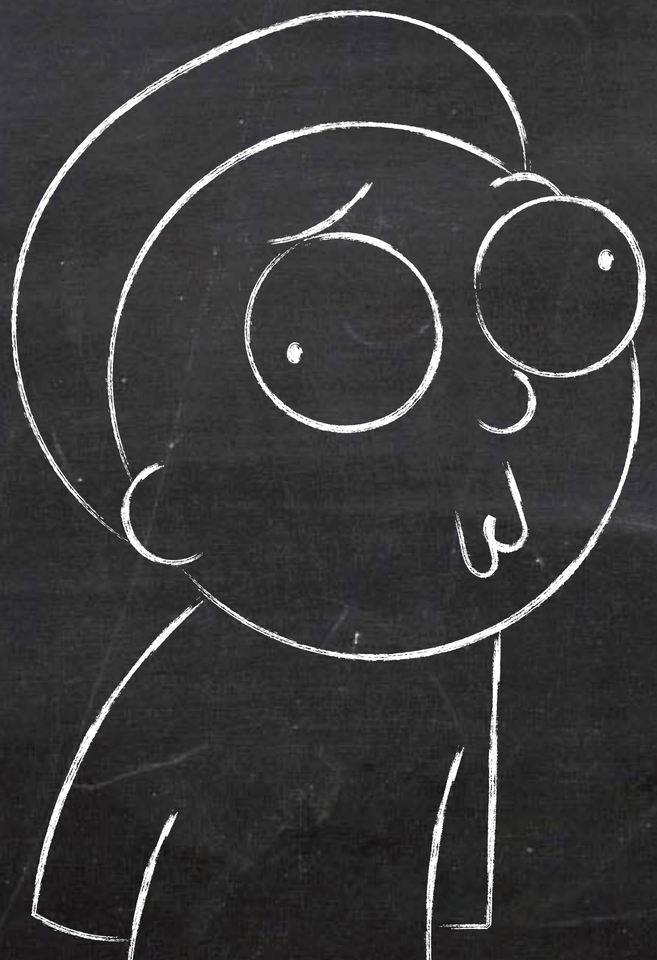
10. `long long` = `long long int` = `signed long long` = `signed long long int` at least [-9'223'372'036'854'775'807, +9'223'372'036'854'775'807]
11. `unsigned long long` = `unsigned long long int` at least [0, +18'446'744'073'709'551'615]

12. `float` single-precision
13. `double` double-precision
14. `long double` usually extended-precision

to get the actual number of bytes used by a particular `type`, use `sizeof(type)`

Overview

basic types



If you keep the following as basic kit in mind, you are fine in this lecture

basic types

1. `char` for bytes

= 8bit
2. `int8_t`
3. `uint8_t`

for (un-)signed tiny integer ranges

= 16bit
4. `int16_t`
5. `uint16_t`

for (un-)signed small integer ranges

= 32bit
6. `int32_t`
7. `uint32_t`

for (un-)signed normal integer ranges

= 64bit
8. `int64_t`
9. `uint64_t`

for (un-)signed huge integer ranges

defined in standard library

`#include <stdint.h>`

10. `size_t` for array indexing, looping on 64-bit machines

defined in standard library

11. `float` for single-precision
12. `double` for double-precision

Types

A type is used to state how a particular bunch of binary data should be interpreted.

```
01 A3 12 F1 04 55 12 AF
```

enumerated types

enumerated types - a type whose value is one of a particular set of *constant values*.

constant values in C:

- integers, decimal represented constants, e.g., `23`
  - octal represented constants prefixed with „`0`“, `027`$_{oct}$ for 23_{dec}
  - hexadecimal represented constants prefixed with „`0x`“, `0x17`$_{hex}$ for 23_{dec}
- longs with suffix „`l`“ resp. „`L`“, e.g., `2300000L`
  - also applies to octal and hexadecimal representation ,e.g., `0x17L`
- character constants is an integer value in character set
  - written in single quote, e.g., `'x'` for 120_{dec} in ASCII or directly as this value

Types

A type is used to state how a particular bunch of binary data should be interpreted.

enumerated types

```
01 A3 12 F1 04 55 12 AF
```

enumerated types - a type whose value is one of a particular set of constant values.

constant values in C:

- string constants are given by double quotes
  - e.g., `"Hello World"`
- floating points with decimal point `42.0` or exponent `1e-2`
  - for float constants, use suffix „`f`“ resp. „`F`“, e.g., `42.0f`
  - for double constants, use suffix „`d`“ resp. „`D`“, e.g., `42.0d`

Types

A type is used to state how a particular bunch of binary data should be interpreted.

enumerated types

```
01 A3 12 F1 04 55 12 AF
```

enumerated types - a type whose value is one of a particular set of *constant values*.

constant values in C:

- list of constant *integer* values identified by particular names

```
enum identifier { enumerator-list }
```

unless defined as a particular type („typedef'd", see later)

the name *identifier* is declared as *tag* for that enumeration type. The *type name* is „`enum identifier`" *not* „`identifier`", e.g.,

`enum compas_direction_e` or
`enum office_rooms_e`

a comma separated list of names with auto-increasing integer value starting at 0 (if value is not explicitly stated), e.g.,
`NORTH, SOUTH, WEST, EAST` or
`OFFICE_ROOM_PINNECKE = 125, OFFICE_ROOM_SAAKE = 110`

Notes on Enumeration Naming, Conventions and Style

more importantly: it must be consistent in the code base!

tag naming

tag cases

```
enum MyEnum
```
(pascal case)

```
enum myEnum
```
(camel case)

```
enum my_enum
```
(snake case) ✗

```
enum MYENUM
```

```
enum MY_ENUM
```

prefixes and suffixes

```
enum prefixtagsuffix
```

(none)	(none)
modul name, e.g. gecko_ ✗	as type, e.g., _t
(others)	as enum, e.g., _e ✗
	(others)

enumeration constants

constant cases

```
DAY_MON, DAY_TUE, DAY_WED, …
```
(all-capital & underscore case)

```
day_mon, day_tue, day_we, …
```
(snake case) ✗

prefixes (if any)

```
// enum my_enum
me_day_mon, me_day_tue, …
```
(abbrev. of tag as prefix) ✗

```
gecko_day_mon, gecko_day_tue, …
```
(module name as prefix)

formatting

```
enum
gecko_days_e { d_mon,
```

```
enum gecko_days_e {
    d_mon, d_tue,...
}
```
✗

```
enum gecko_days_e
{
    d_mon, d_tue,...
}
```

```
enum gecko_days_e
{
    d_mon,
    d_tue,
    ...
}
```

...

✗ convention in this lecture

Notes on Enumeration Naming, Conventions and Style (2)

more importantly: it must be consistent in the code base!

convention, naming and style are context-sensitive!

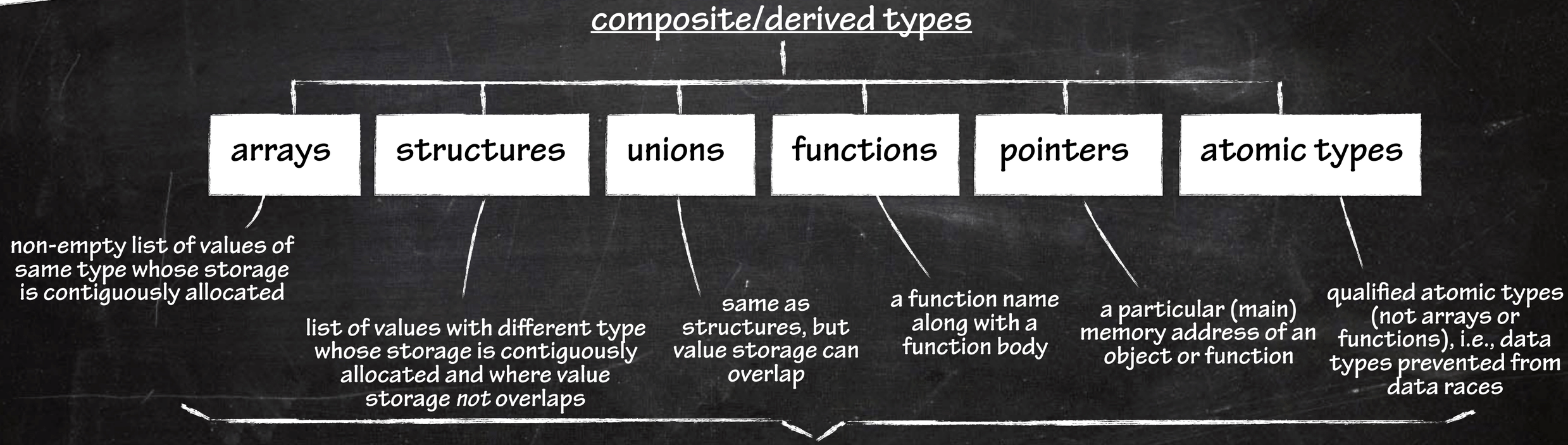
- per-organization policies
  - GNU, see https://www.gnu.org/prep/standards/html_node/Writing-C.html
  - Apache, see https://portals.apache.org/development/code-standards.html
  - NASA, see http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf
  - …
- per-company policies
  - Google, see https://google.github.io/styleguide/cppguide.html
  - Microsoft, see https://msdn.microsoft.com/en-us/library/windows/desktop/aa378932(v=vs.85).aspx
  - …
- per-project policies
  - Linux Kernel, see https://www.kernel.org/doc/html/v4.10/process/coding-style.html
  - GTK+, see https://github.com/GNOME/gtk/blob/master/docs/CODING-STYLE
  - …

Types

A type is used to state how a particular bunch of binary data should be interpreted.

composite/
derived types

```
01 A3 12 F1 04 55 12 AF
```



Implicit Type Conversions (aka implicit casting)

Expressions result in a certain type, and if the expected type differs from the this result type, implicit type conversion typically happens.

```
int sum = 'c' + 42L + 0x123 / 05 - 23.1f * true; // sum is 175
```
example

Implicit type conversion happens by

- by assignments
  - assignment operators `lhs opp rhs` where `opp` is `=`, `+=`, `-=`, ...
  - scalar initializations `= expression` or `= { expression }`
  - function call expressions `expression ( argument-list )`
  - return statement `return expression;` or `return ;`
- for arguments to functions
  - function call expression `int f(int x, int y)` called with `f('c', 43L)`
- use of arithmetic operators
  - conversion to actual type of calculation, e.g., `42 <= 23`
- value transformations, e.g., `long x = 23L; int y = x;`
- array to pointers, function to pointers, ...

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

Explicit Type Conversions (aka explicit casting)

Type conversion explicitly expressed by the programmer. Syntax:

```
(type) expression
```

Example:

```c
int z = 15;

// implicit integer division 1000 by 15 results in 66 which is implicitly casted to float
float x = 1000 / z;

// explicit float division 1000 by 15.0 results in 66.6... which is maps expected type
float y = 1000 / (float) z;

// outputs "x: 66.000000, y: 66.666664"
printf("x: %f, y: %f\n", x, y);
```

Explicit casting might result in loss of information

```c
float pi = 3.14f;
int pii = (int) pi; // pii is exactly three
```

Casting without loss of information:

- integer types to float type
- float types to double type
- character types to integer types

Built-In Type Conversions

Standard library provides built-in functions for type conversions that are not covered by casting:

- string to float
- string to integer
- string to long
- integer to string
- long to string

```
double   atof(const char*)
int      atoi(const char*)
long int atol(const char*)
char    *itoa(int, char*, int)
char    *ltoa(long, char*, int)
```

Objects, Padding and Packing

Note: a C *object* has *no hierarchy* (e.g., inheritance) related other objects. Especially, the term *object* in C is not the same as the term *object* in object-oriented programming.

An *object* is a piece of memory having a particular *value* that is the (type-specific) *interpretation* of that piece of memory.

Objects are created by

- declarations, e.g., `int value;` or `const void *ptr;`
- allocations, e.g., `malloc(20)`
- by string literals, e.g., `"Hello World"`
- ...

Object properties:

- a size in byte, e.g., 4
- a type, e.g., `int`
- a (mutable) value, e.g., 42
- an alignment requirement (see later)
- a storage duration (see later)
- a lifetime
- optionally: a name

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

Objects, Padding and Packing

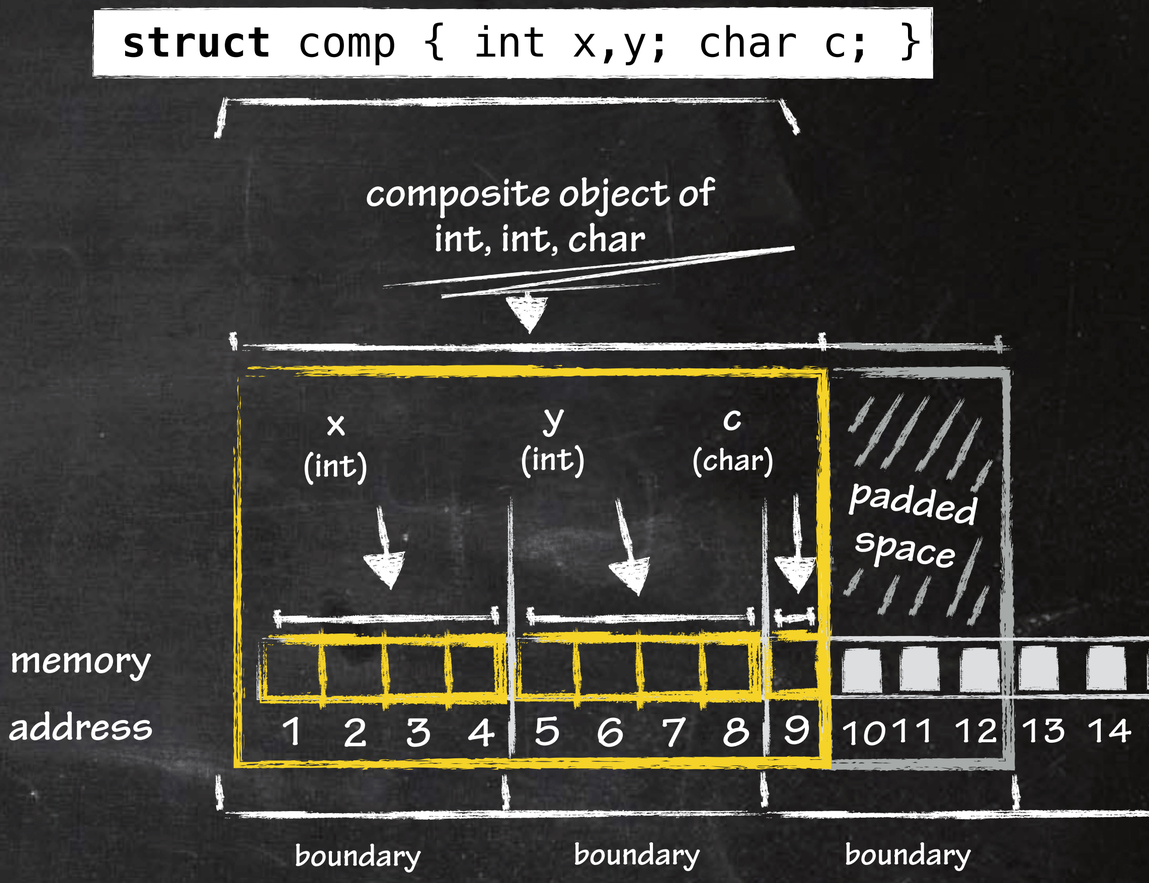
Padding* alignment of composite types to natural address boundaries. May take more space for composite type than actually required but improves memory access performance.

(`sizeof(int)=4` and `sizeof(char)=1`) but

`sizeof(struct comp)=12`

Packing forcing the compiler to *not* apply padding, i.e., composite types has size as expected. May result in slower memory access but better memory usage.

`sizeof(struct comp)=9`



* turned-on by default

Naming, Scoping & Lifetimes

An *identifier* is any sequence of digits, latin letters and underscores that starts with a non-digit character and is *not* a *reserved identifier*. Example: `gecko_tuplet_open` or `result2`

Naming refers to assigning an identifier to one of the following entities:

- objects, e.g., `int value = 7;`
- functions, e.g., `void foo(void)`
- tags, e.g., `enum days { d_mon, d_tue, d_wed }`
- enumeration constants, e.g., `enum days { d_mon, d_tue, d_wed }`
- structure or union members, e.g., `struct { int x, int y, char c }`
- typedef names, e.g., `typedef  long long int64_t`
- goto label names, e.g., `error:`
- preprocessor macro names, e.g., `#define MIN(x, y)`
- preprocessor macro parameter names, e.g., `#define MIN(x, y)`

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

Naming, Scoping & Lifetimes

Reserved identifiers are used for non-user-defined naming and play a particular pre-defined role. Using them is either not allowed or result in undefined behavior.

Reserved identifiers (the most popular ones):

- C language keywords, e.g., `for`, `void`, `return` or `enum`
- Pre-processor directives, i.e., everything starting with `#`, e.g., `#define`
- Names reserved for the standard library
  - leading double underscores, e.g., `__foo`
  - leading single underscore and capital letter, e.g., `_Global`
  - function names of the standard library, e.g., `memcpy` or `malloc`
- Names reserved for future use in the standard library
  - particular function names (e.g., `clog2`), prefixes (e.g., `is` without following underscore), type names(e.g., `intXXXX_t`), particular macro names, and enumeration constants

For more information, see source of these information: http://en.cppreference.com/w/c/language/identifier

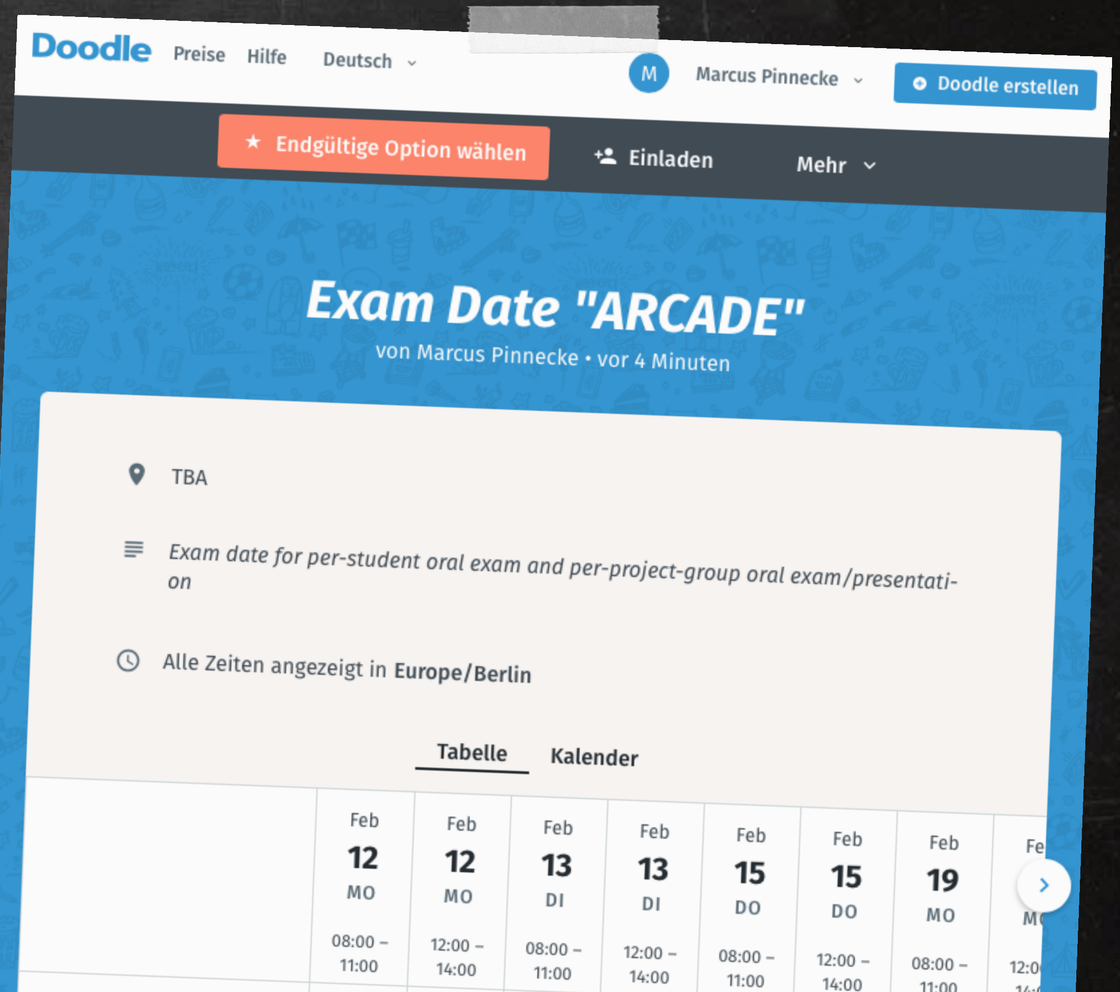
Intermezzo

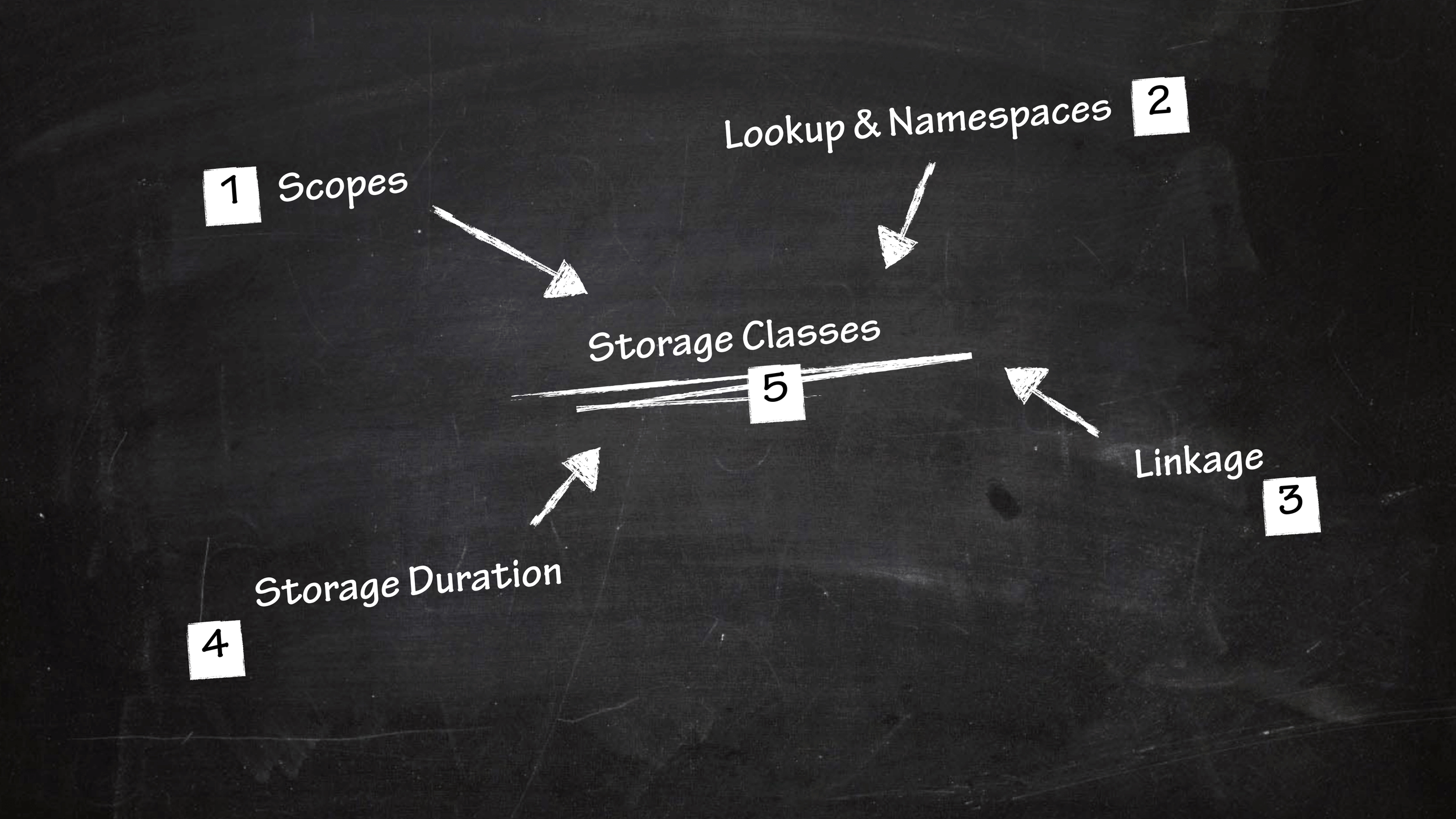
Exam Date

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

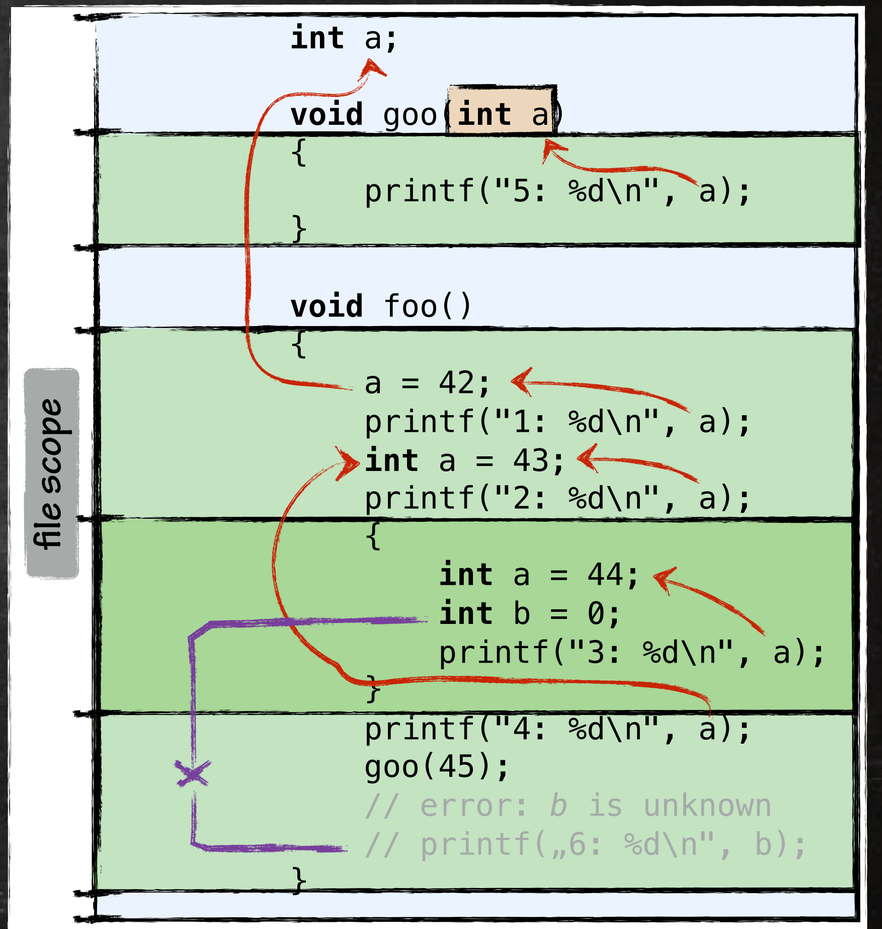
Scopes

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

Scope is 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.

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;
```

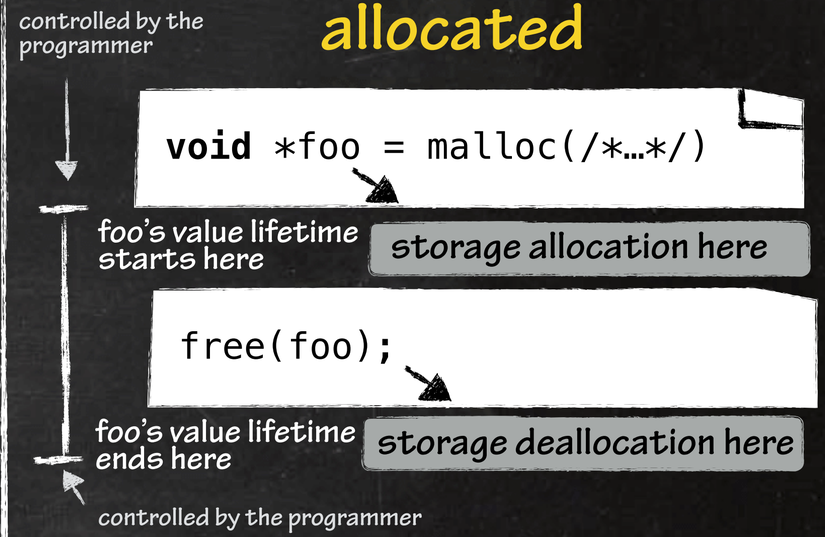
goo.c

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

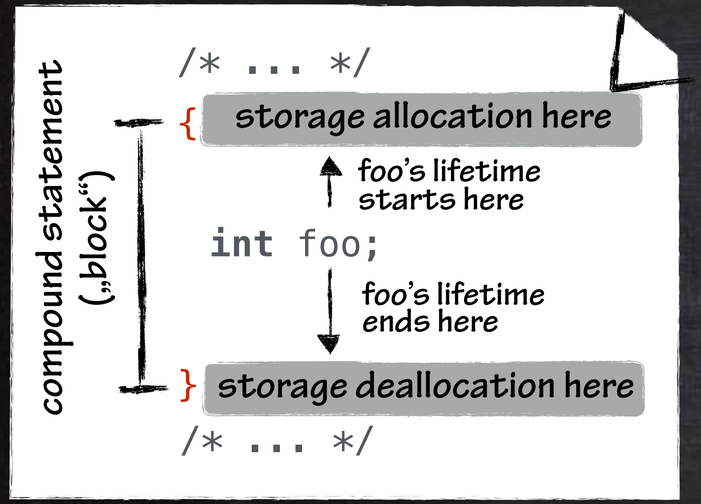
Storage Duration

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

allocated

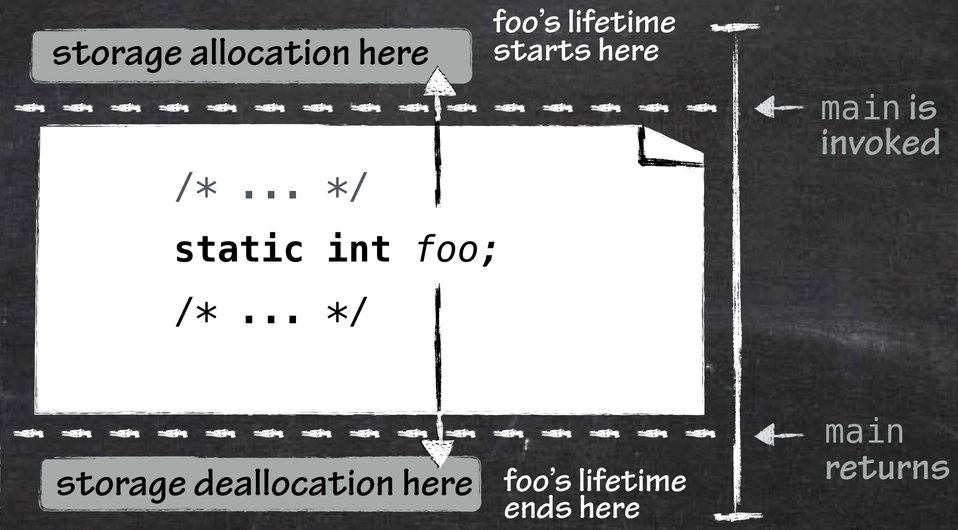


automatic



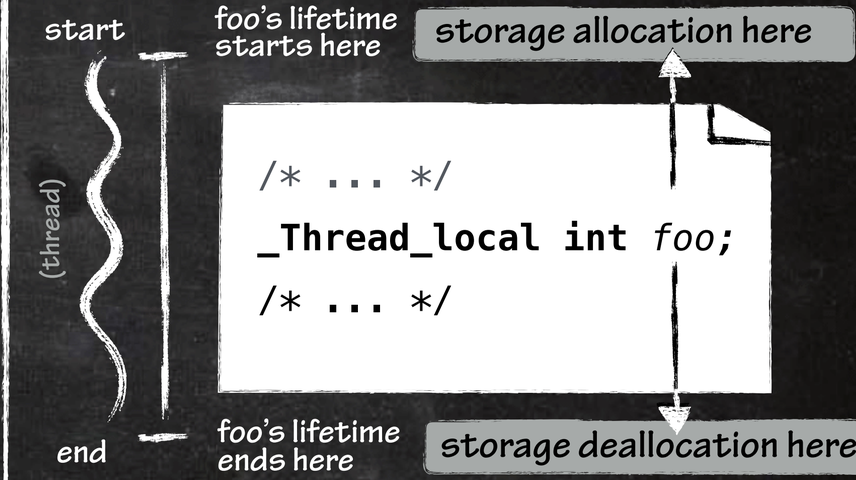
- Object lifetime bound to block

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