Università
della
Svizzera
italiana

Facoltà
di
scienze
informatiche

Edge Computing in the IoT

Recap of C

Luca Butera

C Language

Created in 1972 by Denis Ritchie and Ken Thompson at Bell Laboratories.

Systems programming language

➡ High performance, ease of access to the underlying hardware

➡ Features to build higher-level abstractions and portable programs

First or second most popular programming language every year since 2001 (TIOBE index)

De-facto standard for low-level/high-performance/portable programming

➡ Supported by almost every computing platform, from microcontrollers to supercomputers

Resources

➡ *Systems Programming* course by Prof. Antonio Carzaniga in the BSc Informatics.

➡ C reference on *cppreference.com*

➡ *The C Programming Language, 2nd Edition* by B. W. Kernighan and D. M. Ritchie, 1988.

Modern resources by current members of the C standard committee [both ebooks available free*]:

➡ *Effective C, an introduction to professional C programming* by R. C. Seacord, 2020.

➡ *Modern C* by J. Gustedt, 2019.

Guidelines for writing secure C programs:

➡ *CERT C Coding Standard* by Software Engineering Institute (Carnegie Mellon University), 2016.

*Register on libraries.ch, then use the links above and authenticate as “ETH-Bibliothek (Walk-in)”

Philosophy of C

1. **Trust the programmer:** C assumes you know what you're doing and it lets you.

➡ Which means it also lets you shoot yourself in the foot quite easily!

2. **Keep the language small and simple**

3. **Provide only one way to do an operation**

4. **Make it fast, even if it isn't guaranteed to be portable**

➡ Allowing you to write efficient code is the top priority.

➡ Ensuring that code is portable, safe, and secure is responsibility of the programmer.

R. C. Seacord. *Effective C, an introduction to professional C programming.* 2020.

Program structure

Declaration

```
#include <stdio.h>

int foo();

int main() {
    int a = foo();
    printf("Hello, world! %d\n", a);
    return 0;
}

int foo() {
    return 42;
}
```

Definition

The C compiler behaves as if each file is compiled *top-to-bottom* in a single pass:

➡ Compiler cannot know about elements that come later in the file

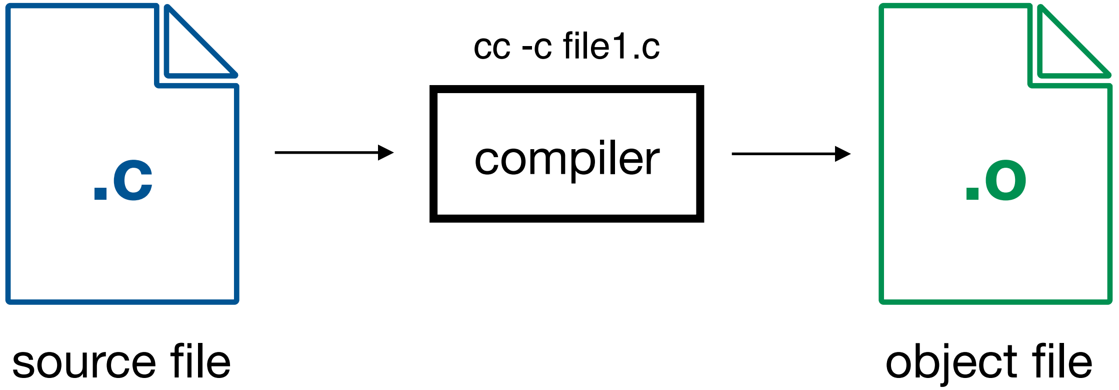
This requires introducing two concepts:

➡ **Declarations** introduce an identifier and describe its type.

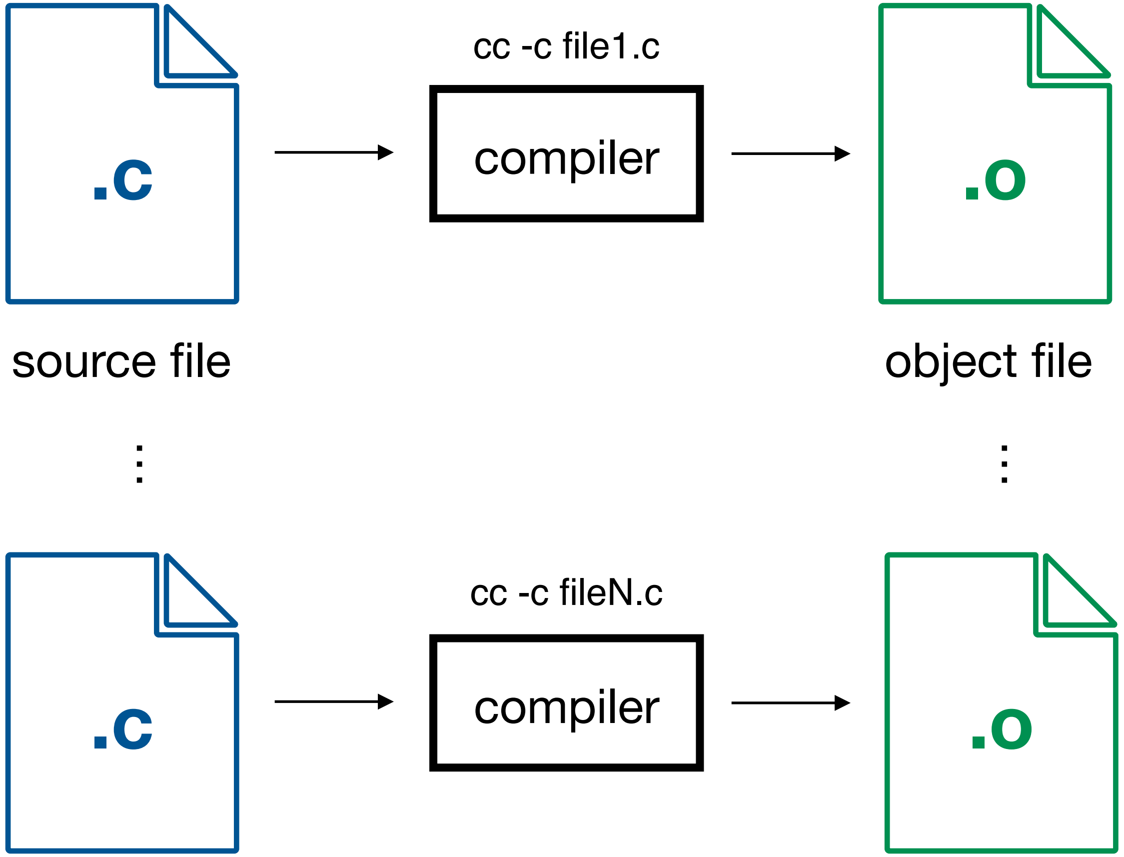
➡ **Definitions** provide the actual implementation.

Fundamental in case of circular references!
i.e. main calls foo and foo calls main

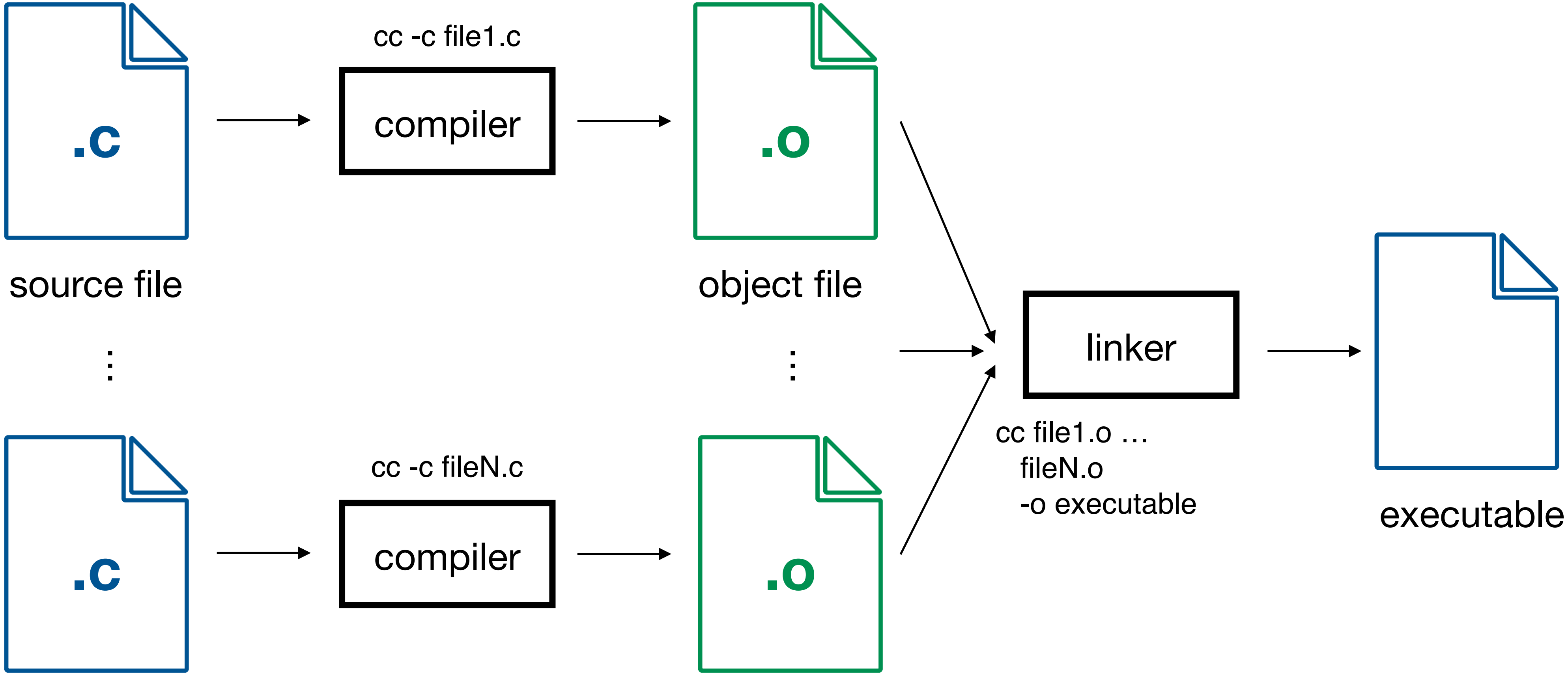
Compilation process



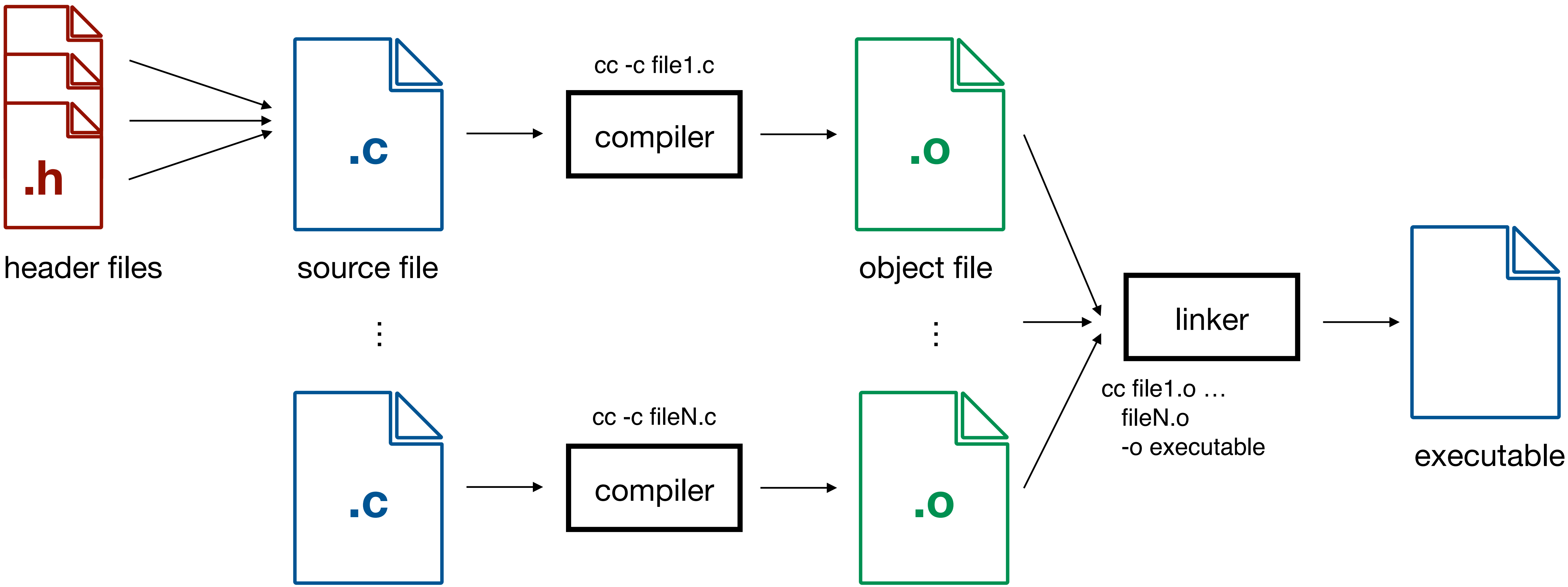
Compilation process



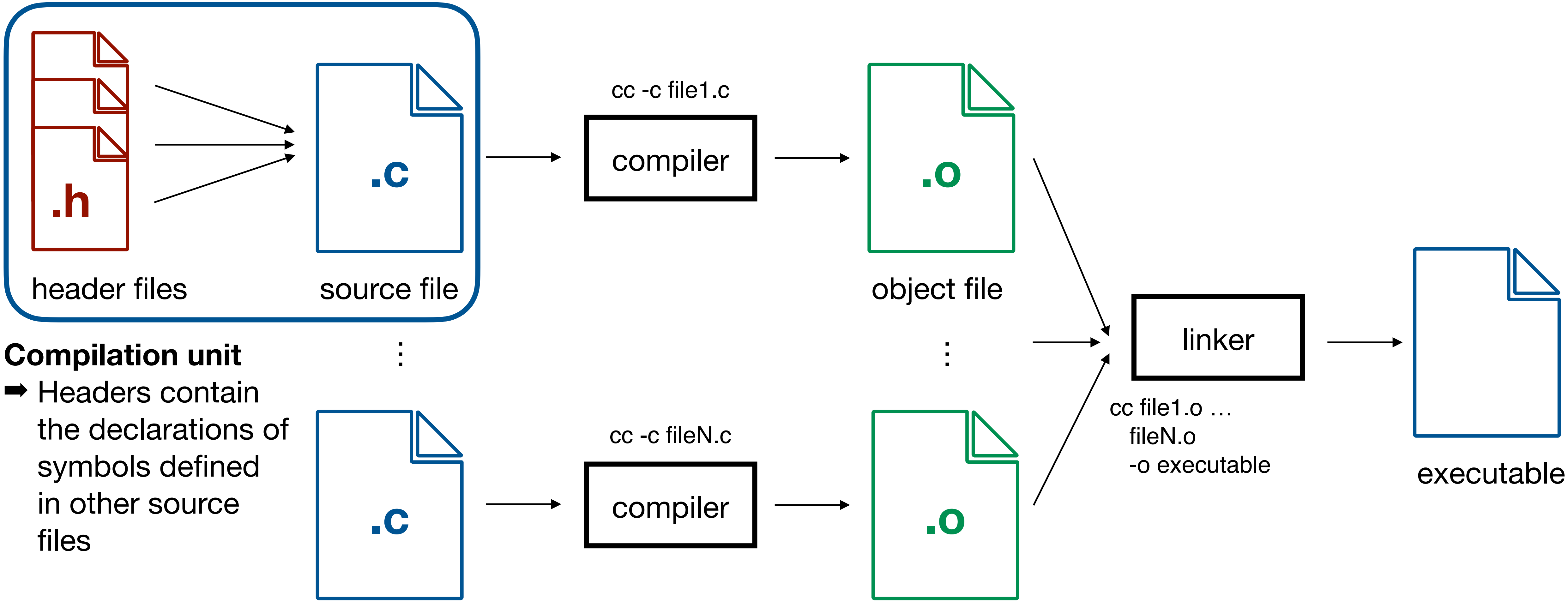
Compilation process



Compilation process



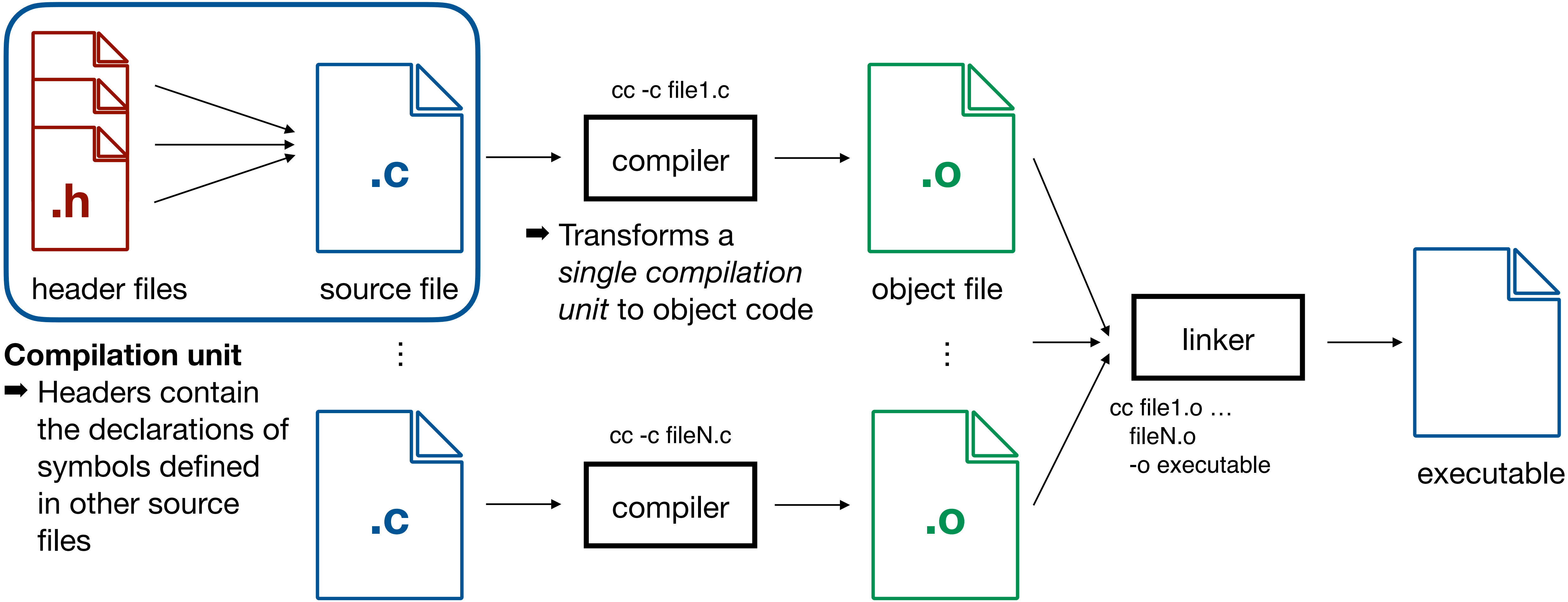
Compilation process



Compilation unit

➡ Headers contain the declarations of symbols defined in other source files

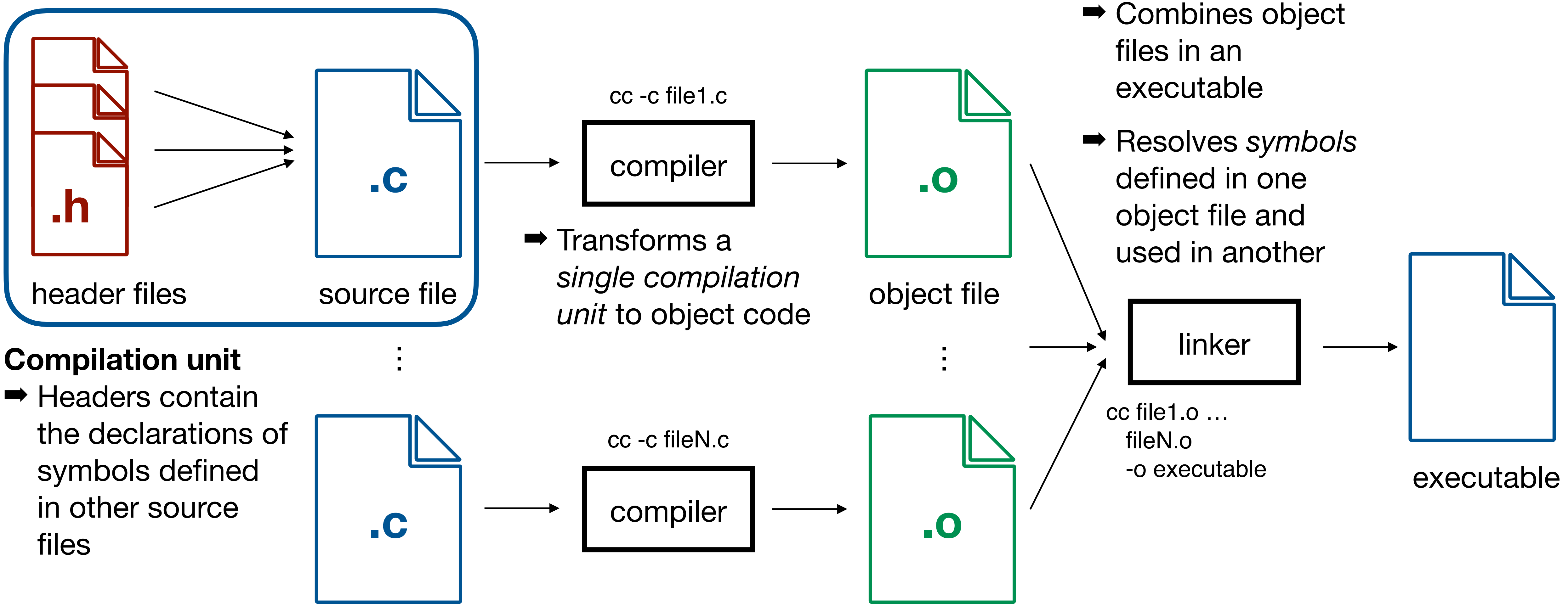
Compilation process



Compilation unit

➡ Headers contain the declarations of symbols defined in other source files

Compilation process



➡ Transforms a *single compilation unit* to object code

➡ Combines object files in an executable

➡ Resolves *symbols* defined in one object file and used in another

Compilation unit

➡ Headers contain the declarations of symbols defined in other source files

Preprocessor

Implements a number of “meta-programming” features on C source files

➡ Result is a single file (the compilation unit) which is passed to the actual compiler.

Preprocessor behavior is controlled by directives, lines that begin with # plus a keyword:

➡ **Macros,** with #define *<MACRO_NAME>* *<body>*

➡ **Conditional compilation,** with #if *<condition>*, #elif *<condition>*, #else and #endif

➡ **Header inclusion,** with #include <…> or #include "…"

Caution: since preprocessor essentially performs text substitution, its behavior can be surprising.
Use it carefully.

Preprocessor

Macros

Replaces every occurrence of the macro's name with the macro's body, potentially with parameters.

```
#define SOME_CONSTANT 4
```

```
#define FUNCTION_MACRO(arg1, arg2) func(arg1, arg2)
```

PITFALL 1: Multiple evaluation of arguments

If an argument appears multiple times in a macro's body, it will be evaluated multiple times, with potential unexpected side effects:

```
#define FOO(a) func(a, a)
```

```
FOO(bar()); // expands to func(bar(), bar()) -> bar() will be called two times
```

Preprocessor

Macros

Replaces every occurrence of the macro's name with the macro's body, potentially with parameters.

#define SOME_CONSTANT 4

#define FUNCTION_MACRO(arg1, arg2) func(arg1, arg2)

PITFALL 2: Multiple statements

Wrap multi-statement macros in do { … } while (false) to ensure they behave as a single standalone statement (i.e. like a normal function call):

#define BAR(…) do { foo(…); bar(…); baz(…); } while (false)

Preprocessor

Macros

Replaces every occurrence of the macro's name with the macro's body, potentially with parameters.

```
#define SOME_CONSTANT 4
```

```
#define FUNCTION_MACRO(arg1, arg2) func(arg1, arg2)
```

PITFALL 3: Definition of variables inside a macro

Only define variables inside a macro when using the do/while form, otherwise the variables will be visible outside the macro:

```
#define FOO(a) do { int _a = a; func(_a, _a); } while (false)
```

```
// This is the correct definition of FOO(a)!
```

Preprocessor

Conditional compilation

Compiles the contents of the first block whose condition evaluates to true.

```
#if !defined(SOME_CONSTANT)
…
#elif SOME_CONSTANT == 4
…
#else
…
#endif
```

PITFALL 4: Inactive blocks are not compiled at all

Compile errors in a conditional compilation block cannot be detected unless the block is currently active.

Preprocessor

Header inclusion

Replaces #include directives with contents of the desired header files, recursively applying the preprocessor on each included file.

#include <system-header.h>

#include "user-header.h"

PITFALL 5: Double inclusion

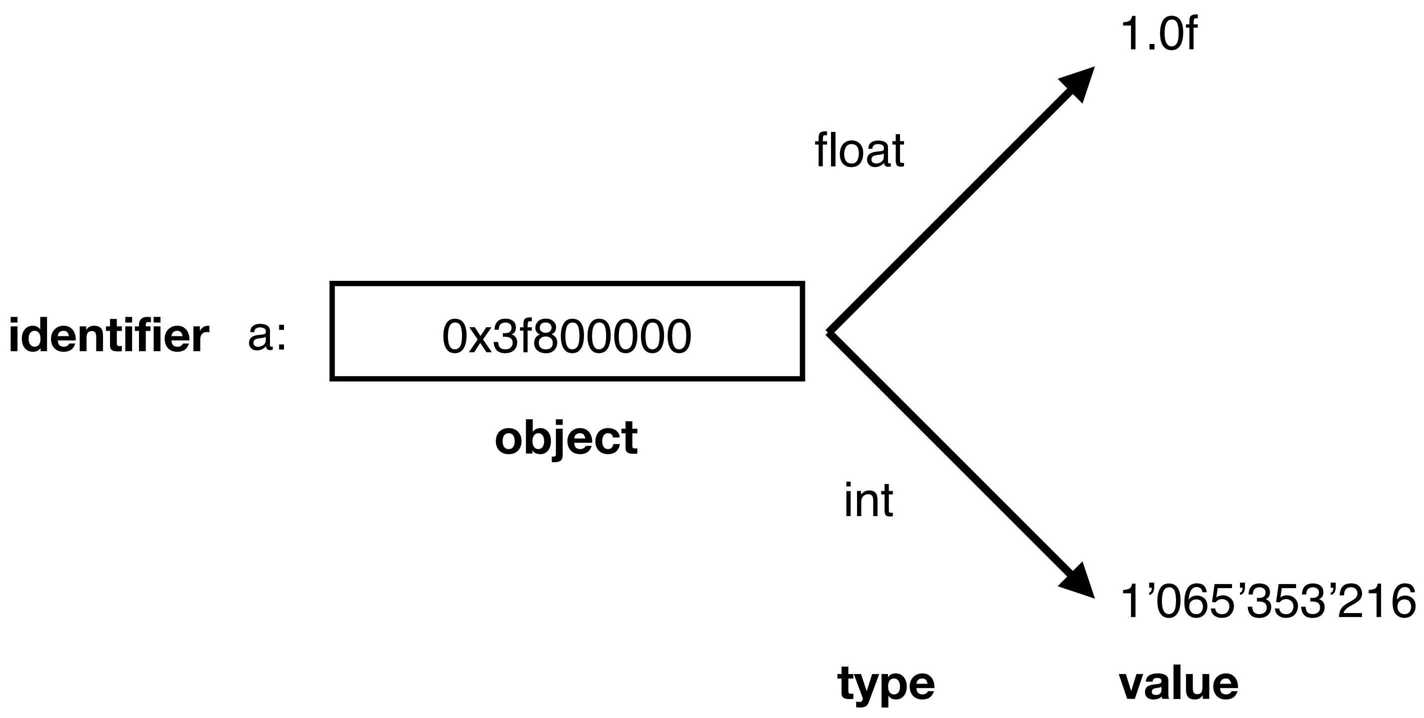
Headers could be included multiple times
(e.g. two headers both include a third header file).

Wrap header files in an *header guard* to prevent unintended side effects:

foo.h:

```
#if !defined(FOO_H)
#define FOO_H
// the actual header file
#endif /* !defined(FOO_H) */
```

Variables



A region of memory is called an **object.**

The contents of an object, when interpreted as a certain **type**, form a **value**.

An object with an associated type and an **identifier** is a **variable:**

float a = 1.0f;

Fundamental Types

Integer types: char short int long long long

➡ The size of each type is *implementation-defined*, most common data models:

ILP32	8	16	32	32	64

32-bit CPUs: int, long and pointers are 32-bit

LP64	8	16	32	64	64

64-bit CPUs (except Win64!): long and pointers are 64-bit, int is still 32-bit

Signedness

➡ Integers types are **signed** by default, the unsigned versions are called **unsigned int**/…

➡ *Exception:* char is implementation-defined, be explicit and use either **signed char** or **unsigned char**

Type promotions

➡ Operation on integers of different sizes implicitly convert all operands to the largest size before computing the result: int + short → int

Fundamental Types

Integer types: char short int long long long

➡ The size of each type is *implementation-defined*, most common data models:

	char	short	int	long	long long
ILP32	8	16	32	32	64
LP64	8	16	32	64	64

ILP32: 32-bit CPUs: int, long and pointers are 32-bit ← Arduino Portenta H7

LP64: 64-bit CPUs (except Win64!): long and pointers are 64-bit, int is still 32-bit

Signedness

➡ Integers types are **signed** by default, the unsigned versions are called **unsigned int**/…

➡ *Exception:* char is implementation-defined, be explicit and use either **signed char** or **unsigned char**

Type promotions

➡ Operation on integers of different sizes implicitly convert all operands to the largest size before computing the result: int + short → int

Fundamental Types

➡ Introduced in C99

Fixed-size integer types: int8_t int16_t int32_t int64_t

➡ Require #include <stdint.h>

➡ Unsigned versions called uint{8,16,32,64}_t

➡ Useful when an explicit data size is desired, e.g. to communicate with other devices

Booleans: bool

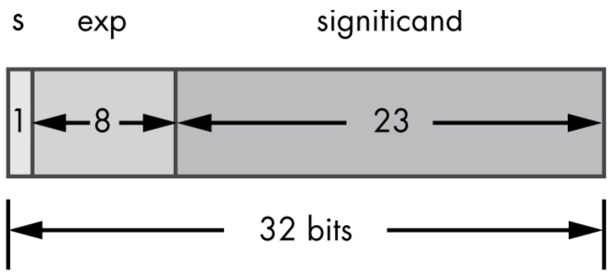
➡ Require #include <stdbool.h>

➡ Only assume values true (1) and false (0), size is 8 bits.

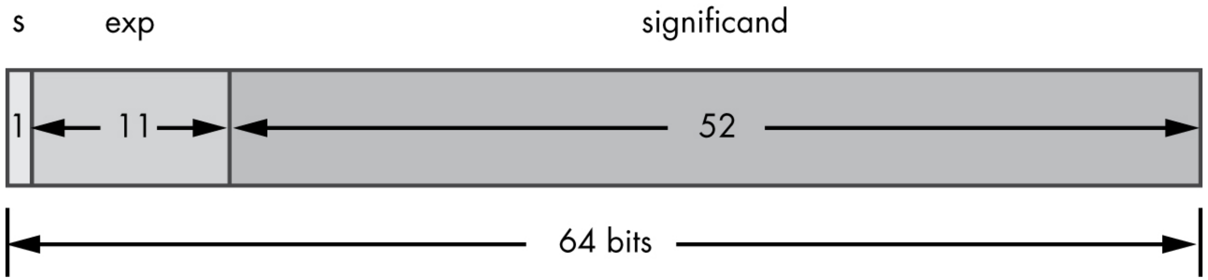
Fundamental Types

Floating-point types: float double

- ➡ Floating point approximates real numbers, storing numbers in binary "scientific" notation: $\pm$ *significand* $\times$ $2^{exponent}$ (e.g. 10.5_{10} -> 1010.1_2 -> $+1.0101 \times 2^3$)
- ➡ Much larger range of representable values, but with approximation errors
- ➡ **Caution:** arithmetic properties (associative, distributive, …) of real numbers, do not hold for floats!
- ➡ Floating point is generally more expensive than integers on embedded systems, small microcontrollers often do not have floating point hardware at all!



float



double

Struct

Define a new type composed of a sequence of *members,* used to group information that is commonly manipulated together.

```
typedef struct {
    int year;
    int month;
    int day;
} date_t;
```

- ➡ Main mechanism for encapsulating data in C programs.
- ➡ Members are laid out consecutively in memory.
- ➡ Structs can be passed to or returned by a function: more convenient than passing members one by one!

Create variable of type date_t and initialize its members:

```
date_t today = {.year = 2021, .month = 10, .day = 12};
```

Accessing members of a struct:

```
printf("%d", today.year);        today.day += 1;
```

Enum

Defines a new type that can represent a **finite** and **known** number of values, called *cases*.

```
typedef enum {
    MONTH_JAN = 1, MONTH_FEB, MONTH_MAR,
    MONTH_APR, MONTH_MAY, MONTH_JUN,
    MONTH_JUL, MONTH_AUG, MONTH_SEP,
    MONTH_OCT, MONTH_NOV, MONTH_DEC
} month_t;

month_t cur_month = MONTH_OCT;
```

➡ Each enum case is represented as an integer value.

➡ Integer values can be specified either explicitly, implicitly or a mix. **Implicitly:** first case has value 0, subsequent cases increase sequentially.

➡ Like with structs, typedef gives a name to the new type created by the enum.

Enums do not constitute a namespace! Good practice: prefix enum cases with the enum's name (i.e. MONTH_*) to limit pollution of the global identifier namespace.

Unions

Unions define a new type that can represent multiple members **alternatively**.

```
typedef union {
    int i;
    float f;
} number_t;

number_t num = {.f = 3.14};
printf("%f\n", num.f); // Prints 3.14
num.i = 123;
printf("%d\n", num.i); // Prints 123
printf("%f\n", num.f); // Undefined behavior!
```

➡ Members all overlap in memory: only one member can be active at a time.

➡ The size of a union is the **size of its largest member**.

➡ Accessing a member that is not active (i.e., not last written) is **undefined behavior**.
You must ensure this doesn't happen!

Putting it together

Example of one pattern that can be implemented using structs, enums and unions

```
typedef struct {
    enum {MESSAGE_SENSOR, MESSAGE_ACTUATOR} type;

    union {
        struct { int room_id; float cur_temp; } sensor;
        struct { int room_id; bool heater; bool cooler; } actuator;
    };
} message_t;
```

Putting it together

Example of one pattern that can be implemented using structs, enums and unions

```
typedef struct {
    enum {MESSAGE_SENSOR, MESSAGE_ACTUATOR} type;

    union {
        struct { int room_id; float cur_temp; } sensor;
        struct { int room_id; bool heater; bool cooler; } actuator;
    };
} message_t;
```

A single struct that can represent different types of messages

Putting it together

Example of one pattern that can be implemented using structs, enums and unions

```
typedef struct {
    enum {MESSAGE_SENSOR, MESSAGE_ACTUATOR} type;

    union {
        struct { int room_id; float cur_temp; } sensor;
        struct { int room_id; bool heater; bool cooler; } actuator;
    };
} message_t;
```

Enum and structs defined inline, at the same time they're used.

Equivalent to defining them beforehand with typedef

A single struct that can represent different types of messages

Putting it together

Example of one pattern that can be in

```
typedef struct {
    enum {MESSAGE_SENSOR, ME

    union {
        struct { int room_id; float cur_temp; } sensor;
        struct { int room_id; bool heater; bool cooler; } actuator;
    };
} message_t;
```

Anonymous union

Members (sensor and actuator) are available directly as members of the outer message_t struct

They still overlap in memory like a normal union!

`message_t msg;` `msg.sensor;` `msg.actuator;`

A single struct that can represent different types of messages

Putting it together

Example of one pattern that can be implemented using structs, enums and unions

```
typedef struct {
    enum {MESSAGE_SENSOR, MESSAGE_ACTUATOR} type;

    union {
        struct { int room_id; float cur_temp; } sensor;
        struct { int room_id; bool heater; bool cooler; } actuator;
    };
} message_t;
```

A single struct that can represent different types of messages

Putting it together

Example of one pattern that can be implemented using structs, enums and unions

```
// Hypothetical function to receive a message
message_t msg = receive_message();

if (msg.type == MESSAGE_SENSOR) {
    printf("Received sensor message: in room %d current temp. is %f",
        msg.sensor.room_id, msg.sensor.cur_temp);
} else if (msg.type == MESSAGE_ACTUATOR) {
    printf("Received actuator message: heater %d, cooler %d in room %d",
        msg.actuator.heater, msg.actuator.cooler, msg.actuator.room_id);
}
```

Variable visibility and lifetime

Scope

Where is the variable declared?

➡ File scope: **Global variables** declared outside a function

➡ Function scope: **Function parameters**

➡ Block scope: **Local variables** declared inside a block (i.e. {…})

Scope determines where a variable is visible: the scope where it is declared and all **inner** ones

Storage duration

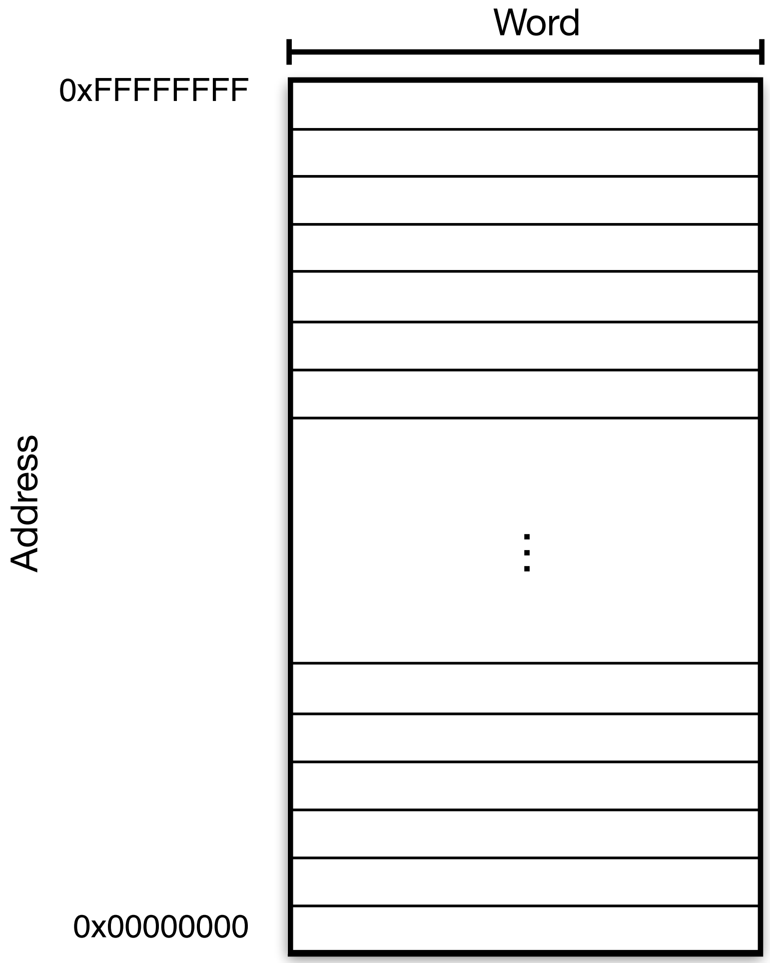
Where is the object stored? How long will it exist?

➡ **Automatic:** local variables
Created when entering their definition scope, destroyed when leaving it

➡ **Static:** global variables, local variables with static modifier
Exist for the entire duration of the program

➡ **Allocated:** dynamic memory allocations
Created with malloc and destroyed with free

Memory model



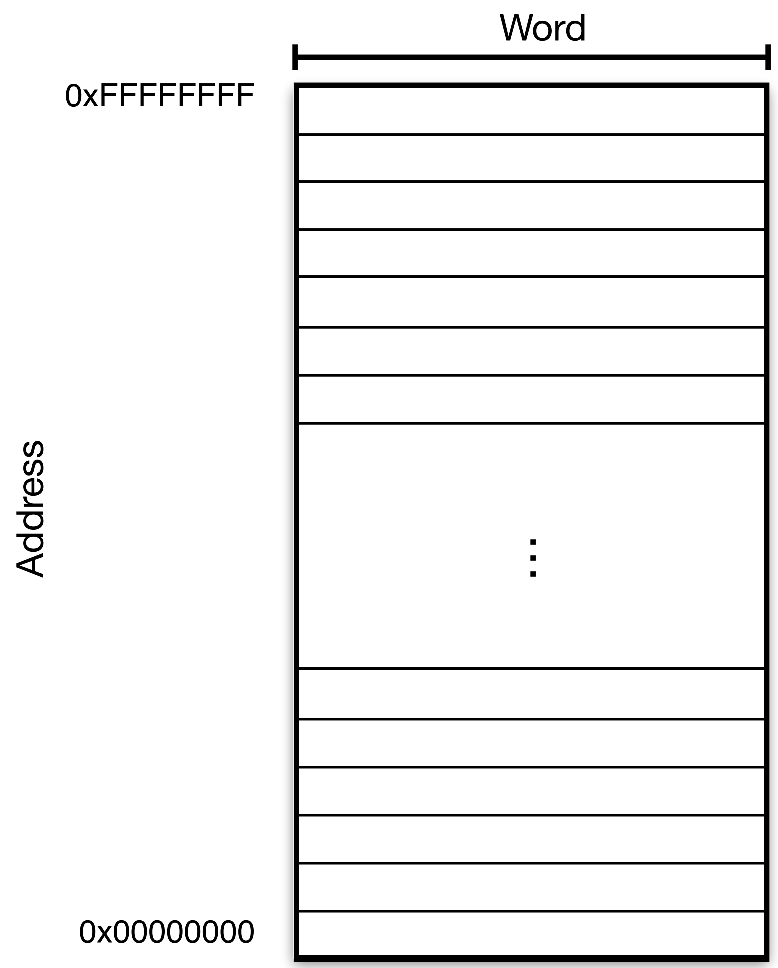
Address: Uniquely identifies memory locations, generally with granularity of one byte

Word: the natural unit of data handled by processor

➡ Word size depends on the specific processor, i.e. 32-bit or 64-bit CPUs. (but small 8-bit microcontrollers are still common in embedded systems!)

➡ Word-sized memory accesses are usually the most efficient, especially if *aligned,* i.e. address is a multiple of word size.

Memory model



Programs see one linear, contiguous address space

➡ An abstraction: not all memory is created equal

➡ You only see this when working at a very low level, but it's useful to know

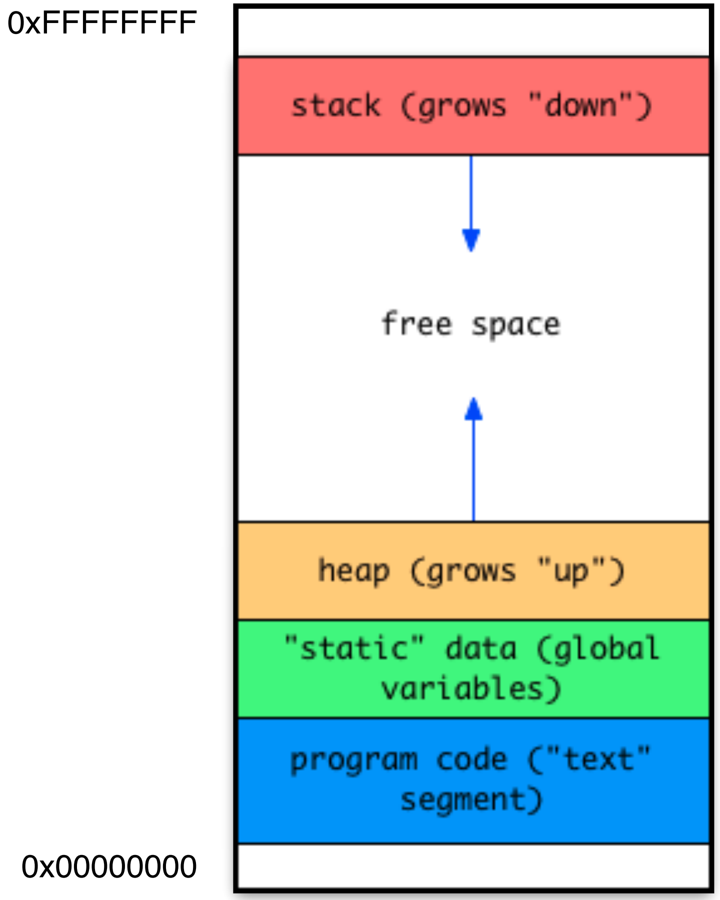
Embedded systems

➡ multiple types of memories: nonvolatile Flash, on-chip SRAM, off-chip DRAM

➡ memory-mapped input/output: configure and communicate with external peripherals by reading/writing special memory regions.

➡ …

Memory model



Memory is divided in “areas” with different purposes. At least 4 are present in C programs on all platforms:

Local variables defined inside a function (automatic storage duration)

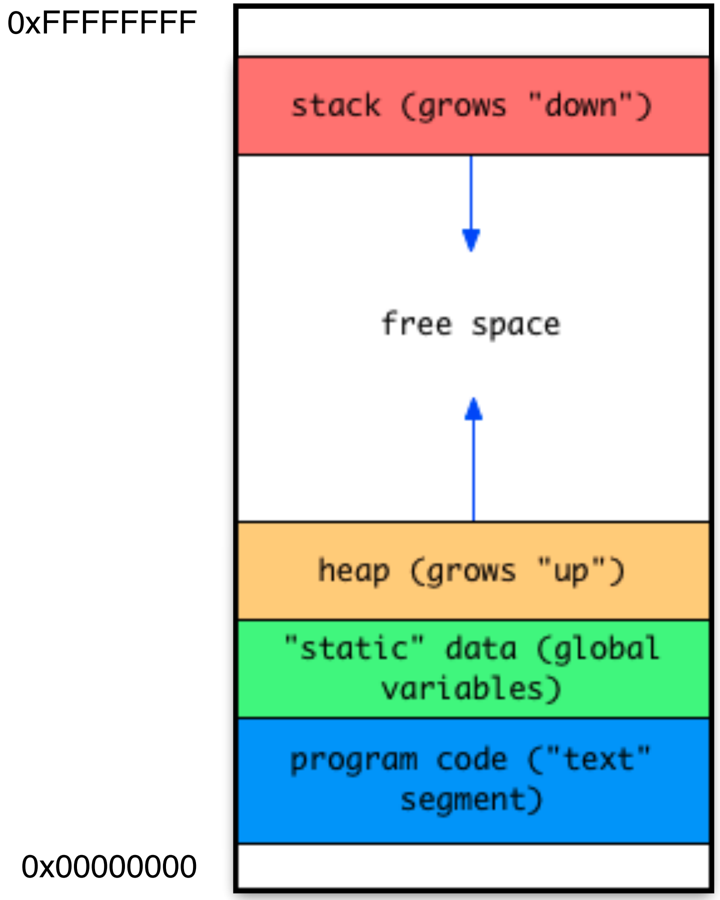
Dynamic memory allocations managed with malloc/free (allocated storage duration)

Global variables defined outside a function or
Local variables defined inside a function with the static modifier (static storage duration)

Executable code

https://jsommers.github.io/cbook/pointersarrays.html

Memory model



Where will the following objects be stored?

int **a** = 1; ← static data

static int **b** = 2;

int main() { ← stack

  int **c** = 2;

  foo();

}

void foo() {

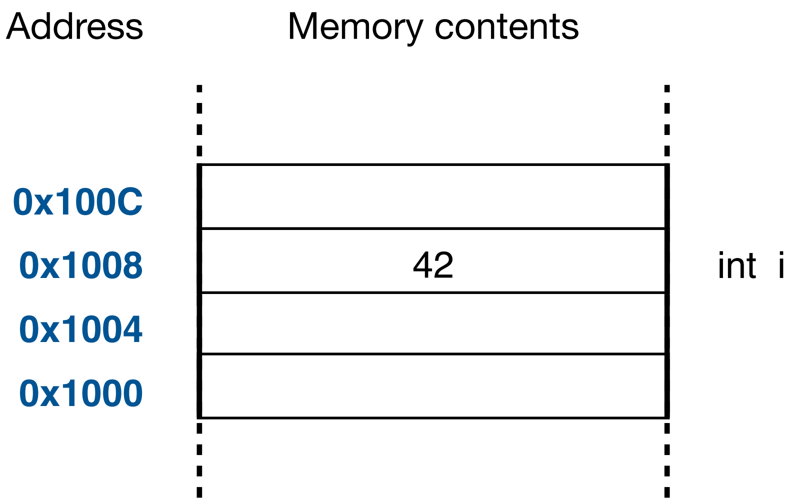
  static int **d** = 3; ← static data

}

https://jsommers.github.io/cbook/pointersarrays.html

Pointers

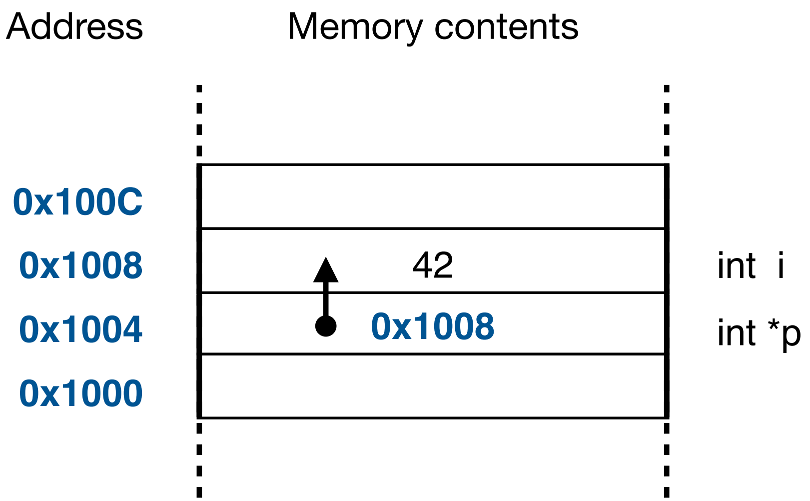
A pointer is an object whose value is the **memory address of another object**.



int i = 42;

Pointers

A pointer is an object whose value is the **memory address of another object**.




```
int  i = 42;

int *p = &i;
```

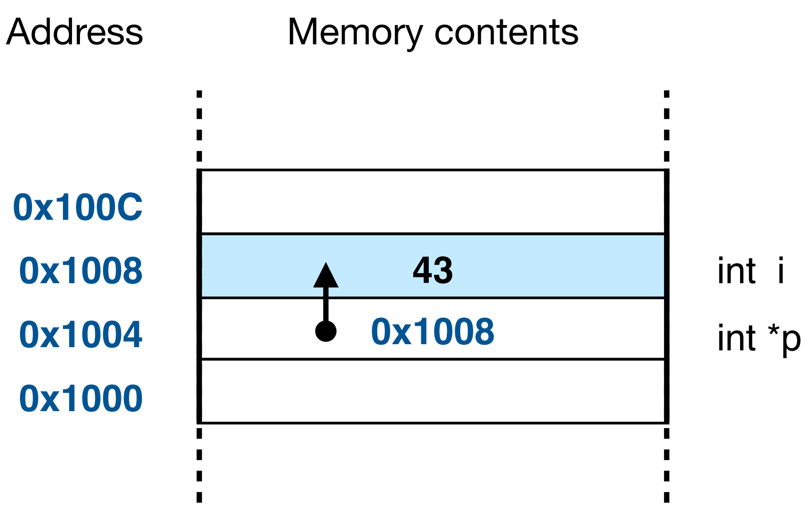
We say that:

➡ int * is a *pointer to* int

➡ p *points to* i

Pointers

A pointer is an object whose value is the **memory address of another object**.




```
int  i = 42;

int *p = &i;

*p = *p + 1;

// i == 43
```

We say that:

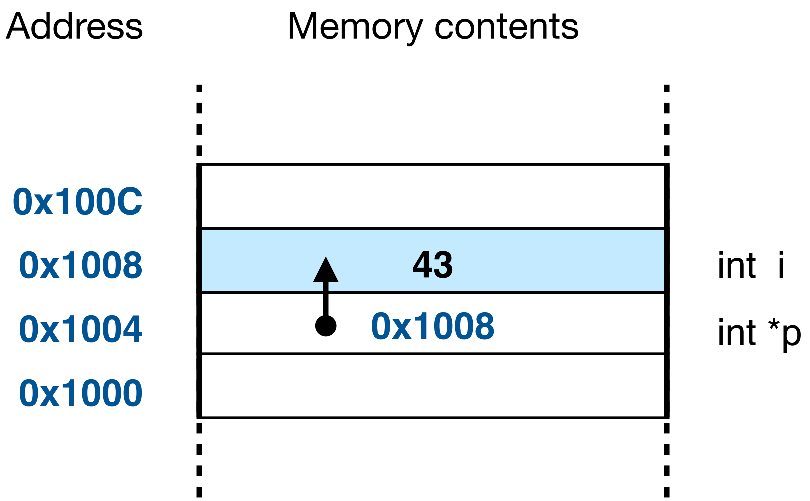
➡ int * is a *pointer to* int

➡ p *points to* i

We can operate on i indirectly, through p

Pointers

A pointer is an object whose value is the **memory address of another object**.




```
int  i = 42;

int *p = &i;

*p = *p + 1;

// i == 43
```

We say that:

➡ int * is a *pointer to* int

➡ p *points to* i

We can operate on i indirectly, through p

Fundamental operations

&i — *Address-of* operator: returns the memory address of i (i.e. 0x1008)

*p — *Dereference* operator: follows the pointer, reading or writing the value of i (i.e. 42)

Pointers

Pointers are typed char *c; // pointer to char int *i; // pointer to int

➡ At runtime, pointers only store a memory address.
Type is fundamental to know the size of the pointed object and how to interpret it.

➡ Casts between pointer types, e.g. (char *)i, reinterpret pointed memory as a different type.

Special case! void *p; // pointer to void

➡ A pointer that can point to *any* object. It must be cast to a concrete type before dereferencing!
int *i = (int *)p; // *i can be dereferenced, *p cannot

➡ Allows to implement functions that manipulate arbitrary types (e.g. malloc/free).

Pointers

Pointer to nothing p = NULL;

➡ Marks that a pointer doesn't point to any valid object. Cannot be dereferenced, but can be checked! if (p == NULL) { // do something }

➡ Commonly used to indicate that an object has not been created yet, that an operation has failed, …

Multiple level of indirection int **p; // pointer to a pointer to an int

➡ Pointers can also point to *other pointers*.

Struct pointers typedef struct { int a; int b } my_struct; my_struct *s;

➡ Accessing members of my_struct through a pointer would require **(*s).a**
*s.a would correspond to *(s.a)

➡ C provides **s->a**, a convenience operator equivalent to (*s).a

Uses of pointers

Build data structures (e.g. linked lists)

In-out function parameters

➡ Pass-by-reference parameters

➡ Storing an output value in memory provided by the caller

➡ Returning multiple outputs

Avoid copying large structs when passed to a function (performance optimization)

typedef struct { int a[100]; } large_struct_t;

void func(large_struct_t arg) {…} -> void func(**large_struct_t** *arg) {…}

Uses of pointers: pass-by-reference

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int x = 1, y = 2;
swap(x, y);
// x is still 1, y is still 2
```

C is *pass-by-value*: swap operates on a copy of the values contained in x and y

Pointers make it possible to implement *pass-by-reference*.

Uses of pointers: pass-by-reference

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int x = 1, y = 2;
swap(x, y);
// x is still 1, y is still 2
```

```c
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int x = 1, y = 2;
swap(&x, &y);
// now x is 2, y is 1
```

C is *pass-by-value*: swap operates on a copy of the values contained in x and y

Pointers make it possible to implement *pass-by-reference*.

Arrays

Arrays represent a **contiguous sequence of objects** that all have the same type.

Creating and initializing an array

float array[4] = {1.0, 2.5, 5.0, 7.5};

In C, arrays always have a **fixed size**, that is specified when the array is created and cannot be changed later.

Subscript operator array[i]: accessing array elements

float a = array[0];   array[0] += 1;

Pointer to array: The array identifier is a pointer to the first element of the array
float *p = array; // equivalent to &array[0]

Pointer arithmetic

Arithmetic operations on pointers:

p + n p - n	p1 - p2	++p --p	p++ p--
sum / subtract an integer	subtract two pointers	pre-increment pre-decrement	post-increment post-decrement

Comparison operations on pointers:

<   <=   ==   >=   >

Pointer arithmetic

Arithmetic operations on pointers:

p + n p - n	p1 - p2	++p --p	p++ p--
sum / subtract an integer	subtract two pointers	pre-increment pre-decrement	post-increment post-decrement

Comparison operations on pointers:

<   <=   ==   >=   >

Pointer arithmetic

Arithmetic operations on pointers:

p + n p - n	p1 - p2	++p --p	p++ p--
sum / subtract an integer	subtract two pointers	pre-increment pre-decrement	post-increment post-decrement

Comparison operations on pointers:

<   <=   ==   >=   >

Pointer arithmetic operates in terms of elements of the **pointer's type**:

int arr[10];   int *p = arr;   ***(p + 2) == arr[2]**

0x1000 → **0x1008**

+2 ints -> +8 bytes

Pointer arithmetic and arrays are very closely related in C!

Dynamically allocated memory

Handle situations in which exact memory requirements are unknown until program is run.

Address

void *malloc(size_t bytes);

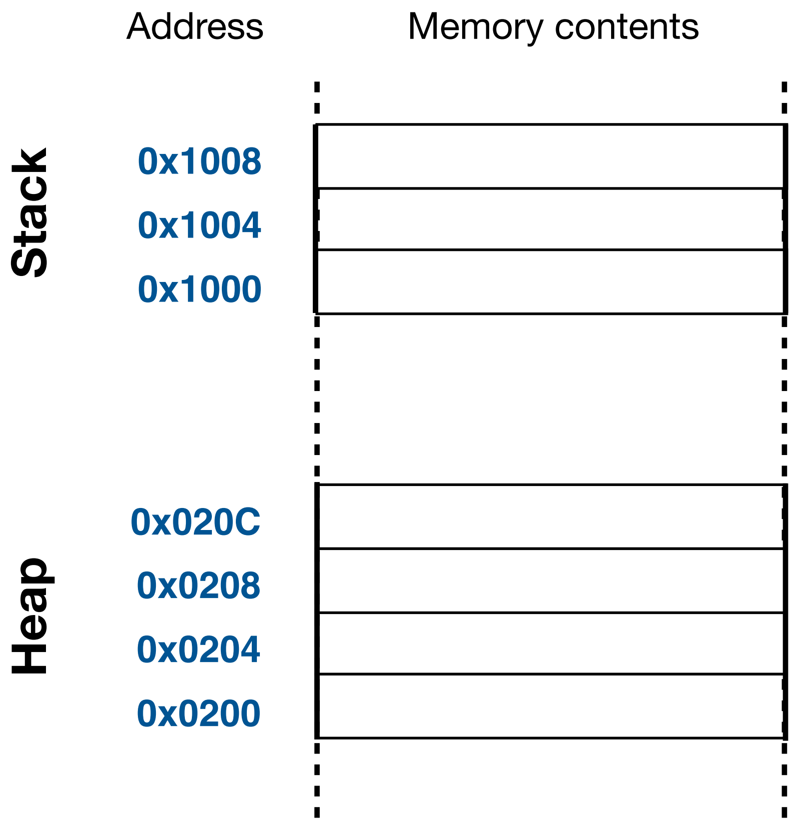
void free(void *p);

Usage

```
int *array = (int *)malloc(4 * sizeof(int));
if (array == NULL) {
    // Allocation failed!
}
// Use our array of integers…
free(array);
```

Dynamically allocated memory

Handle situations in which exact memory requirements are unknown until program is run.



void *malloc(size_t bytes);

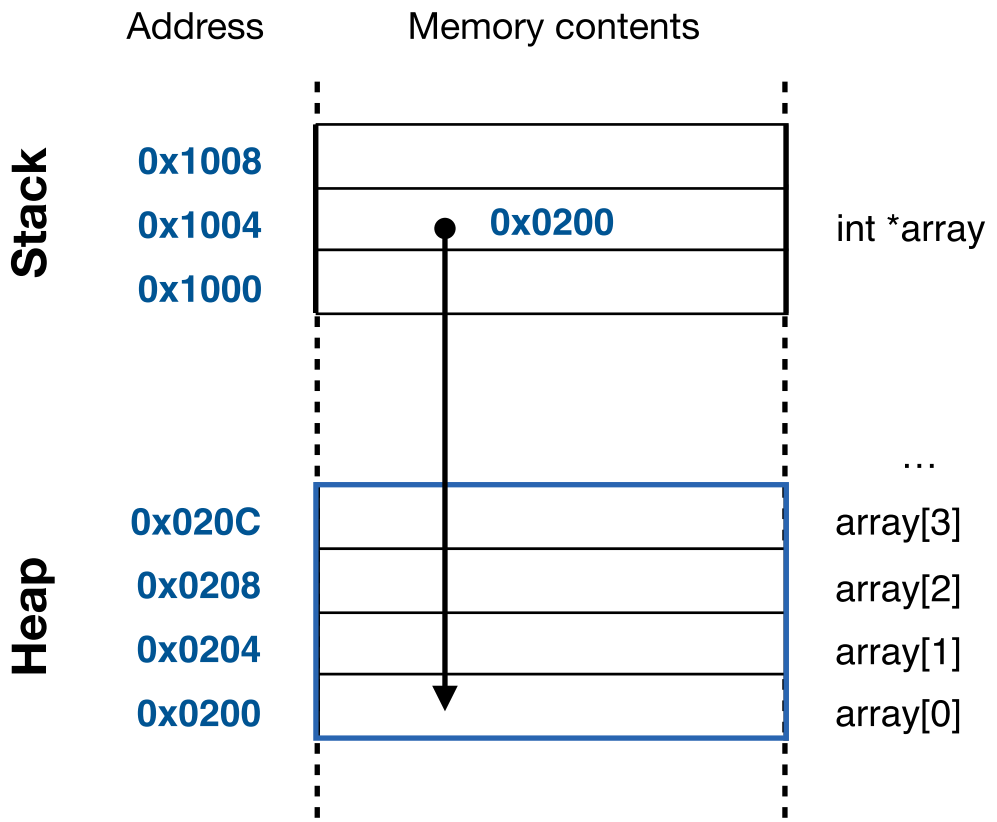
void free(void *p);

Usage

```
int *array = (int *)malloc(4 * sizeof(int));
if (array == NULL) {
    // Allocation failed!
}
// Use our array of integers…
free(array);
```

Dynamically allocated memory

Handle situations in which exact memory requirements are unknown until program is run.



void *malloc(size_t bytes);

void free(void *p);

Usage

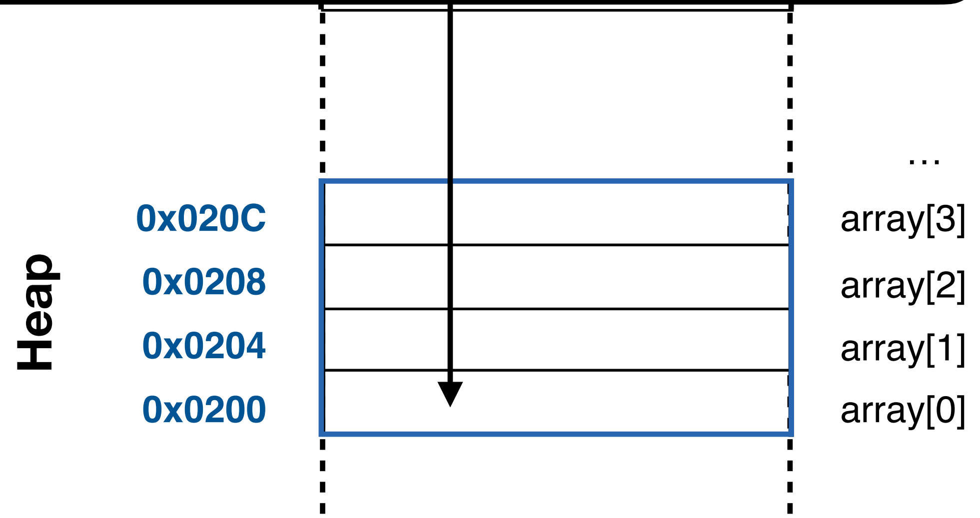
```
int *array = (int *)malloc(4 * sizeof(int));
if (array == NULL) {
    // Allocation failed!
}
// Use our array of integers…
free(array);
```

Dynamically allocated memory

Handle situations in which exact memory requirements are unknown until program is run.

malloc and free use **void *** to remain agnostic w.r.t contents of the allocated memory.

You can cast the returned pointer to any type



void *malloc(size_t bytes);

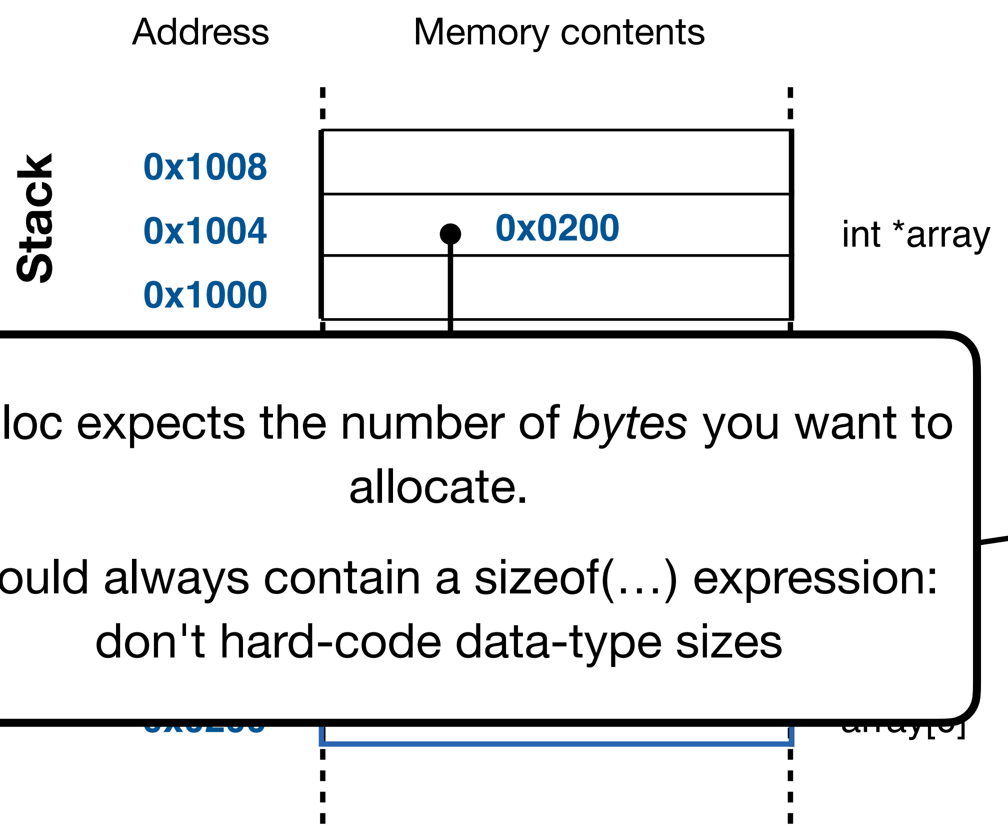
void free(void *p);

Usage

```
int *array = (int *)malloc(4 * sizeof(int));
if (array == NULL) {
        // Allocation failed!
}
// Use our array of integers…
free(array);
```

Dynamically allocated memory

Handle situations in which exact memory requirements are unknown until program is run.



malloc expects the number of *bytes* you want to allocate.

Should always contain a sizeof(…) expression: don't hard-code data-type sizes

void *malloc(size_t bytes);

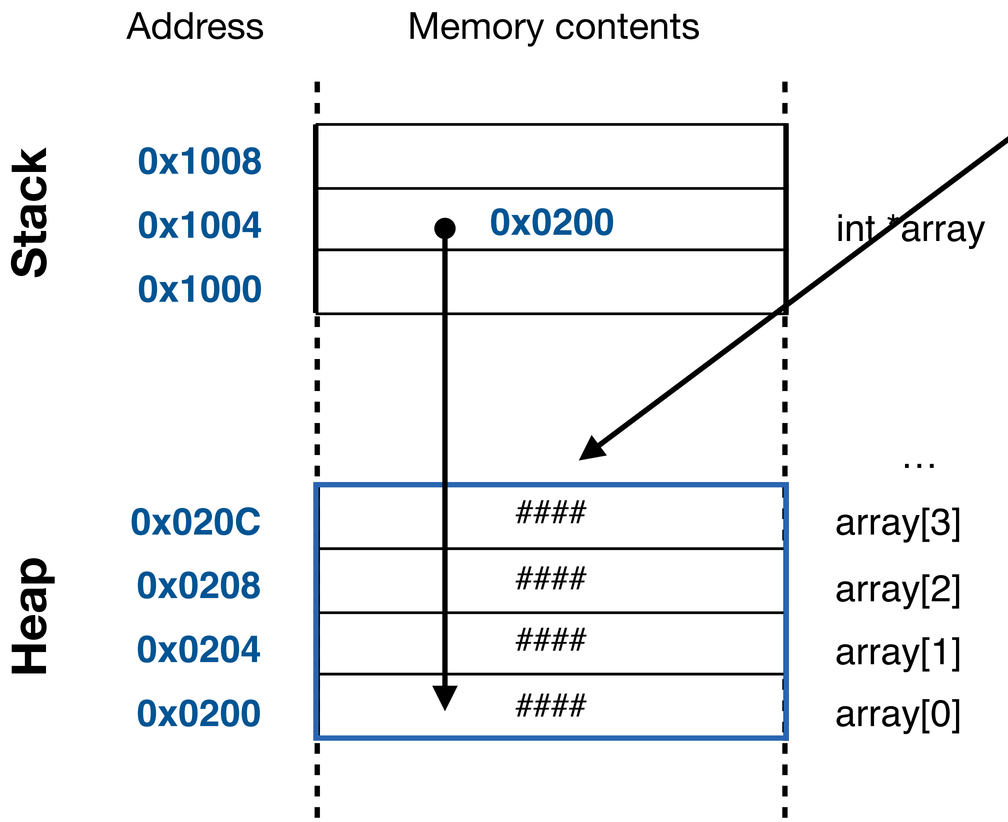
void free(void *p);

Usage

```
int *array = (int *)malloc(4 * sizeof(int));
if (array == NULL) {
        // Allocation failed!
}
// Use our array of integers…
free(array);
```

Dynamically allocated memory

Handle situations in which exact memory requirements are unknown until program is run



Memory returned by malloc is uninitialized!

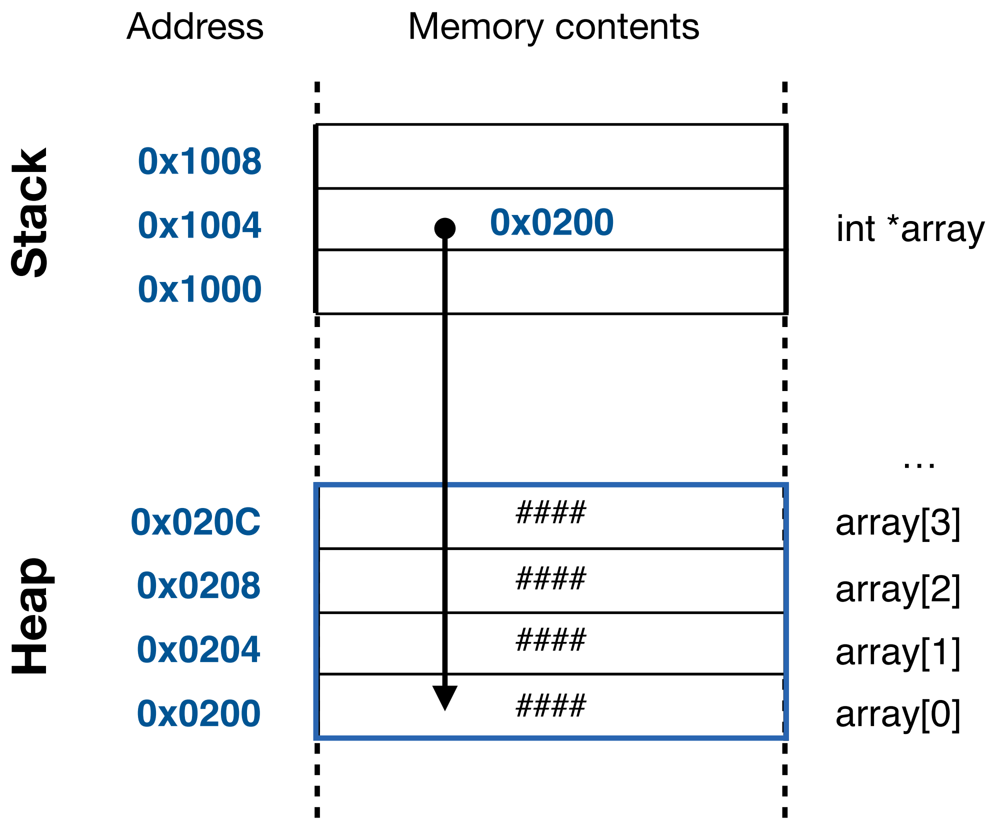
You must explicitly initialize it (assign a value) before use

Usage

```
int *array = (int *)malloc(4 * sizeof(int));
if (array == NULL) {
    // Allocation failed!
}
// Use our array of integers…
free(array);
```

Dynamically allocated memory

Handle situations in which exact memory requirements are unknown until program is run.



void *malloc(size_t bytes);

void free(void *p);

Usage

```
int *array = (int *)malloc(4 * sizeof(int));
if (array == NULL) {
	// Allocation failed!
}
// Use our array of integers…
free(array);
```

Bitwise operators

~a	a & b	a | b	a ^ b
bitwise NOT	bitwise AND	bitwise OR	bitwise XOR (exclusive OR)

Boolean operations on *individual bits* of variables (typically unsigned int)

NOT

~	0	1	=
	1	0	

Output bit is **negation** of input

AND

0	0	1	1	&
0	1	0	1	=
0	0	0	**1**	

Output bit is 1 if **both** inputs were 1

OR

0	0	1	1	|
0	1	0	1	=
0	**1**	**1**	**1**	

Output bit is 1 if **at least one** input was 1

XOR

0	0	1	1	^
0	1	0	1	=
0	**1**	**1**	0	

Output bit is 1 if **exactly one** input was 1 (in other words, if inputs were different)

Bitwise operators

a **<<** n	a **>>** n
bitwise shift left	bitwise shift right

Shift left

1	0	1	1	0	0	1	0	**<<** 2	=
1	1	0	0	1	0	**0**	**0**		

Move bits to the left by 2 positions.
Bits introduced on the right are always **zero**

Shift right

1	0	1	1	0	0	1	0	**>>** 3	=
0	**0**	**0**	1	0	1	1	0		

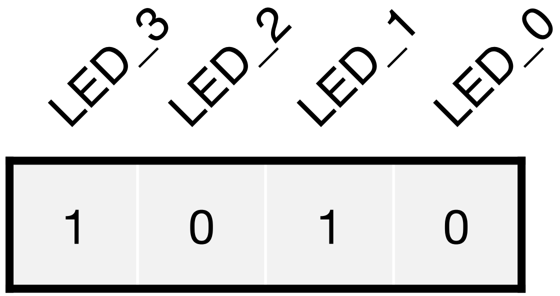
Move bits to the right by 3 positions.
Bits introduced on the left are:

➡ always **zero** for unsigned types

➡ equal to the **sign** for signed types (sign extension)

Uses of bitwise operators: Bitmasks

A bitmask is a way to manage a set of **Boolean flags** in a single integer variable



State of 4 LEDs

```
typedef enum {
    LED_0 = (1 << 0),  // 1 = 0001
    LED_1 = (1 << 1),  // 2 = 0010
    LED_2 = (1 << 2),  // 4 = 0100
    LED_3 = (1 << 3),  // 8 = 1000
} led_state_t;

led_state_t leds = 0;  // All LEDs off
```

Enable LED_0

`leds |= LED_0;`

Clear LED_1

`leds &= ~LED_1;`

Toggle LED_2

`leds ^= LED_2;`

Read LED_3

`if (leds & LED_3) {…}`

Uses of bitwise operators: Performance

Some maths operations involving **powers of two** can be implemented with bitwise operators, which are faster to execute on hardware:

➡ **Powers of 2:** 2^n == 1 << n

➡ **Multiplication:** x * 2^n == x << n

➡ **Division:** x / 2^n == x >> n

➡ **Modulo:** x % 2^n == x & (1 << n)

Usually compilers can optimize these operations automatically **but only if n is a constant!** Else, you need to explicitly use bitwise operators if you want the performance win.

Università
della
Svizzera
italiana

Facoltà
di
scienze
informatiche

Edge Computing in the IoT

Recap of C++

Luca Butera

C++ Language

Originally created in 1979 by Bjarne Stroustrup at Bell Laboratories as "C with Classes"

Goal: an extension to C that helped large-scale software development

Modern C++ has become a very versatile language:

➡ Support for all major programming paradigms (procedural, object-oriented, functional)

➡ High control over memory and other system resources

➡ Focus on performance, cross-platform portability

Versatility is paid with a significantly more complex language compared to C.

Note: C++ is not strictly a superset of C
Enough differences prevent compiling C programs as C++ as-is, without changes.

Resources

- ➡ *Systems Programming* course by Prof. Antonio Carzaniga in the BSc Informatics.
- ➡ C++ reference on *cppreference.com*

Books

- ➡ *The C++ Programming Language, 4th Edition* by B. Stroustrup, 2013.
- ➡ *Effective Modern C++* by S. Meyers, 2014 [ebook available free*].

*Register on libraries.ch, then use the links above and authenticate as "ETH-Bibliothek (Walk-in)"

Object-oriented programming

Main addition of C++ compared to C

Objects (in OOP sense) are a mechanism to encapsulate data and code

➡ Similarities with structs in C? **YES**

OOP in C++ extends structs with member functions, access modifiers, inheritance, …

	struct		class
	struct box {		**class** box {
Data members	double width;	≈	double width;
	double height;		double height;
	double depth;		double depth;
Member functions	double get_volume();		double get_volume();
	};		};

Object-oriented programming

Mai
Obj   d code

➡ S
  O   odifiers, inheritance, …

struct and class are almost indistinguishable in C++,

except for the member's default access specifier

class is the preferred form

```
struct box {
    double width;
    double height;
    double depth;

    double get_volume();
};
```

≈

```
class box {
    double width;
    double height;
    double depth;

    double get_volume();
};
```

Data members

Member functions

Member functions

Implement functionality that operates on data members managed by an object

Declaration in box.h

```
class box {
    /* … */

    double get_volume();
};
```

Definition in box.cpp

```
double box::get_volume() {
    return width * height * depth;
}
```

Usage

```
box box1 = {.width = 10, .height = 20, .depth = 5};
double volume = box1.get_volume();
std::cout << "Volume of the box is: " << volume << std::endl;
```

Static members

Data and function members can be declared static, to share them among all class instances

Declaration in box.h

```
class box {
    /* … */

    static int FACES;
};
```

Definition in box.cpp

```
static int box::FACES = 6;
```

Usage

```
std::cout << "Boxes have << box::FACES << " faces." << std::endl;
```

Access specifiers

Information hiding: objects should expose a stable interface, while hiding internal implementation details from the rest of the program.

Access specifiers control which members of an object are accessible from the rest of the program:

- ➡ public: members are accessible anywhere (public interface)
- ➡ protected: members are accessible only inside derived classes (derived class interface)
- ➡ private: members are accessible only inside the class (implementation details)

Access specifiers

```
struct box {
    // public by default
    double width;
    double height;
    double depth;

    double get_volume();
};
```

```
class box {
    // private by default
    double width;
    double height;
    double depth;

    double get_volume();
};
```

Rationale: structs are public-by-default for compatibility with C (which has no access specifiers)

Access specifiers

```cpp
struct box {
private:
    double width;
    double height;
    double depth;

public:
    double get_volume();
};
```

=

```cpp
class box {
// private by default
    double width;
    double height;
    double depth;

public:
    double get_volume();
};
```

class is preferred because private-by-default is safer:
you must explicitly mark members as public to make them accessible

Overloading

C++ allows to define multiple functions with the **same name** but **different parameters**

In calls to overloaded functions, compiler selects the best matching definition: *overload resolution*

➡ Number and type of arguments to the function are considered

➡ Return type is **NOT** considered

```
void print(float f) {
    std::cout << "Float: " << f << std::endl;
}

void print(const char *s) {
    std::cout << "String: " << s << std::endl;
}
```

Usage

```
print(3.14);
// Float: 3.14

print("Hello, world!");
// String: Hello, world!
```

Constructor

Special member function that runs at object creation:
Opportunity to initialize data members and potentially run other code

Declaration in box.h

```
class box {
    box(double width, double height, double depth);
};
```

Definition in box.cpp

```
box::box(double _width, double _height, double _depth)
    : width(_width), height(_height), depth(_depth) {
    // Other initialization code
}
```

Constructor

Special member function that runs at object creation:
Opportunity to initialize data members and potentially run other code

Declaration in box.h

```
class box {
    box(double width, double height,
};
```

Member initialization list:

Provides initialization values for each data member.
Why not use a normal assignment?

Definition in box.cpp

```
box::box(double _width, double _height, double _depth)
    : width(_width), height(_height), depth(_depth) {
    // Other initialization code
}
```

Destructor

Special member function that runs at object destruction:
Opportunity to perform cleanup of resources, free memory, …

Declaration in box.h

```
class box {
   ~box();
};
```

Definition in box.cpp

```
box::~box() {
   // Clean up code
}
```

Usage:

```
int main() {
      box box1(10, 20, 5); // Invokes the three-parameter constructor
      // Use box…
      // box1 goes out of scope, compiler automatically invokes destructor
}
```

Copy constructor

Constructor automatically invoked by compiler every time a copy of an object is needed

Declaration in box.h

```
class box {
    box(const box &other);
};
```

Definition in box.cpp

```
box::box(const box &other)
    : width(other.width),
      height(other.height),
      depth(other.depth) {}
```

Usages

```
box box1(10, 20, 5);
box box2(box1); // Explicit
box2 = box1;    // Assignment
```

```
// Function params ↓
void some_func(box arg) {
        return arg; // Function return
}
```

Copy construc

Constructor automatica ... **ded**

New type: reference to box

Very similar to a pointer, but requires no explicit *address-of* (&) and *dereference* (*) operators

Declaration in box.h

An implicit copy constructor is generated by the compiler, if one is not provided by the programmer

Default implementation is essentially equivalent to this one.

Explicit copy constructor used to implement more advanced behavior, e.g. classes that manage dynamic object allocations.

Definition in box.cpp

```
box::box(const box &other)
  : width(other.width),
    height(other.height),
    depth(other.depth) {}
```

Usages

```
box box1(10, 20, 5);
box box2(box1); // Explicit
box2 = box1;    // Assignment
```

```
// Function params ↓
void some_func(box arg) {
    return arg; // Function return
}
```

Dynamic memory allocations

Objects in C++ are allocated on the stack by default. Like in C, heap allocations must be manually managed.

```
int main() {
    // Allocated on the stack, automatically destroyed when leaving scope.
    box box1(10, 20, 5);

    // Dynamically allocated on the heap and initialized using 3-param constructor.
    box *box2 = new box(20, 30, 50);

    // Access members using arrow -> operator.
    double vol = box2->get_volume();

    ...

    // Destroy object and release memory with delete.
    delete box2;
}
```

Dynamic memory allocations

Objects in C++

Unlike C, new and delete combine memory allocation with initialization.

This ensures that constructors and destructors are always called properly.

```
int main() {
    // Allocated
    box box1(10, 20, 5);

    // Dynamically allocated on the heap and initialized using 3-param constructor.
    box *box2 = new box(20, 30, 50);

    // Access members using arrow -> operator.
    double vol = box2->get_volume();

    ...

    // Destroy object and release memory with delete.
    delete box2;
}
```

Dynamic memory allocations: arrays

Dynamic arrays are allocated and destroyed with special new [] and delete [] syntax

```
int main() {
    // Dynamically allocated array.
    box *box_array = new box[10];

    // Access members using arrow -> operator.
    double vol = box_array[3].get_volume();

    ...

    // Must be released with delete [].
    delete [] box_array;

}
```

Inheritance

C++ supports inheritance to share code and model relationships between classes

Base class

```
class polygon {
public:
    int get_sides();
};
```

Derived classes

class rectangle **: public polygon** { … };

class triangle **: public polygon** { … };

Public and protected members of the base class are accessible as members of derived classes:

```
rectangle rect;        rect.get_sides();
```

C++ also supports *multiple* inheritance: a derived class can inherit from multiple base classes at the same time.

Inheritance

C++ supports inherita

Inheritance access specifier puts a cap on accessibility of inherited members

- ➡ public inheritance: inherited members retain their original access
- ➡ protected inheritance: public members of the base class inherited as protected members on derived classes
- ➡ private inheritance: inherited members accessible as private

Base class

class polygon {

public:

  int get_sides();

};

Derived classes

class rectangle **: public polygon** { … };

class triangle **: public polygon** { … };

Public and protected members of the base class are accessible as members of derived classes:

rectangle rect; rect.get_sides();

C++ also supports *multiple* inheritance: a derived class can inherit from multiple base classes at the same time.

Polymorphism

In object-oriented programming, polymorphism allows to pass derived classes where a base class would be expected.

➡ Derived classes can override the behavior of the base class

Base class

```
class polygon {
    …

public:
    double get_area() { return 0.0 }
};
```

Derived classes

```
class rectangle : public polygon {
    double width, height;

public:
    double get_area() {
        return width * height;
    }
};
```

Polymorphism

Polymorphism is possible only when operating on objects through a pointer

Usage

```
polygon *poly;
rectangle rec {.width = 10, .height = 7};
triangle  tri {.width = 5, .height = 3};

// store address of rectangle and get rectangle area.
poly = &rec;
poly->get_area();

// store address of triangle and get triangle area.
poly = &tri;
poly->get_area();
```

Polymorphism

Polymorphism is possible only when operating on objects through a pointer

Usage

pol

rec

tria

Both will return 0.0. Do you know why?

// store address of rectangle and get rectangle area.

poly = &rec;

poly->get_area();

// store address of triangle and get triangle area.

poly = &tri;

poly->get_area();

Polymorphism

Polymorphism is possible only when operating on objects through a pointer

Usage

```
polygon *poly;
rectangle rec {.width = 10, .height = 7};
triang

// sto
poly =
poly->get_area();

// store address of triangle and get triangle area.
poly = &tri;
poly->get_area();
```

By defaults member functions are *resolved statically* at compile time,

the compiler only looks at the static type of the pointer variable.

Polymorphism

To get the desired behavior, the member function must be declared virtual in the base class

Rationale: dynamic resolution has a cost at runtime, since every function call must look-up what implementation should be called. Must explicitly opt in!

Base class

```
class polygon {
    …

public:
    virtual double get_area() {
        return 0.0
    }
};
```

Derived classes

```
class rectangle : public polygon {
    double width, height;

public:
    double get_area() {
        return width * height;
    }
};
```

Polymorphism

To get the desired behavior, the member function must be declared virtual in the base class

Rationale: dynamic resolution has a cost at runtime, since every function call must look-up what implementation should be called. Must explicitly opt in!

You can also mark the derived member functions as virtual

This allows further derived classes to inherit from rectangle and override get_area().

```
public:
    virtual double get_area() {
        return 0.0
    }
};
```

```
    double width, height;

public:
    double get_area() {
        return width * height;
    }
};
```

Polymorphism

Pure virtual function: declare a function that every derived class must implement

(equivalent to abstract methods in Java)

Base class

```
class polygon {
  …

public:
  virtual double get_area() = 0;
};
```

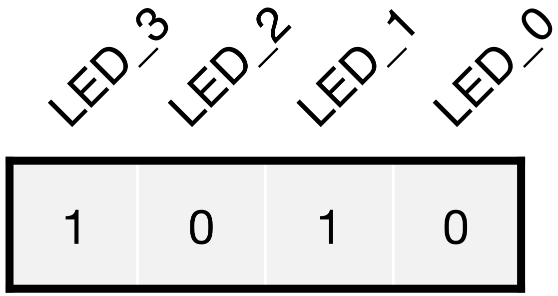
Derived classes

```
class rectangle : public polygon {
  double width, height;

public:
  double get_area() {
    return width * height;
  }
};
```

Bitmasks with bitwise operators

A bitmask is a way to manage a set of **Boolean flags** in a single integer variable



State of 4 LEDs

```
typedef enum {
    LED_0 = (1 << 0),  // 1 = 0b0001
    LED_1 = (1 << 1),  // 2 = 0b0010
    LED_2 = (1 << 2),  // 4 = 0b0100
    LED_3 = (1 << 3),  // 8 = 0b1000
} led_state_t;

led_state_t leds = 0;  // All LEDs off
```

Enable LED_0

```
leds |= LED_0;
```

Clear LED_1

```
leds &= ~LED_1;
```

Toggle LED_2

```
leds ^= LED_2;
```

Read LED_3

```
if (leds & LED_3) {…}
```

Bit fields

C and C++ both support defining struct members with explicit size in bits

```
struct bit_field_t {
  // <type> <name> : <size>;
     unsigned int a : 3;
     unsigned int b : 2;

     // Force to start a new byte
     unsigned int   : 0;

        bool c : 1;
};
```

Adjacent bit fields usually share and straddle bytes

➡ Cannot take address of a bit field

Limited supported types for bitfields in C:

➡ unsigned int and signed int

➡ bool (with #include <stdbool.h>)

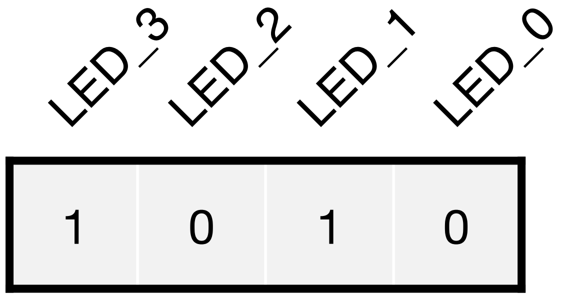
C++ additionally support:

➡ Any signed or unsigned integer type (char, short, …)

➡ enums

Bitmasks with bit fields

A bitmask is a way to manage a set of **Boolean flags** in a single integer variable



State of 4 LEDs

```
typedef struct {
    unsigned int led_0 : 1;
    unsigned int led_1 : 1;
    unsigned int led_2 : 1;
    unsigned int led_3 : 1;
} led_state_t;

led_state_t leds = {0};  // All LEDs off
```

Enable LED_0

`leds.led_0 = 1;`

Clear LED_1

`leds.led_1 = 0;`

Toggle LED_2

`leds.led_2 ^= 1;`

Read LED_3

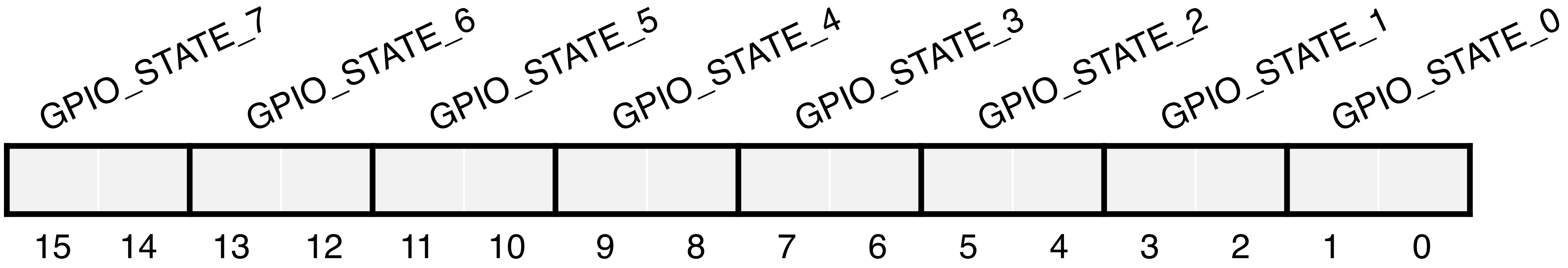
`if (leds.led_3) {…}`

Questions?

Exercises

Exercise 1

An hypothetical microcontroller has the following hardware register to configure 8 GPIO pins



Each GPIO_STATE_# field might assume one of four values:

00 input pin, pull-down   01 input pin, pull-up   01 output pin   11 reserved

➡ Define code to set/get/clear each field of this register with enums, bitwise operators and macros

➡ Repeat the exercise using bit fields

Note: for this exercise, you can assume the register is a normal unsigned short global variable

What are the pros and cons of each approach?

Exercise 2

Complete the following statements using pointers and pointer arithmetic

```
int main() {
    int var = 42;
    // Save the address of var to a pointer variable called "varptr"
    <...>

    // Complete the following statements using varptr
    printf("var is located at address: %p\n", <...>);
    printf("varptr points to a location that contains: %d\n", <...>);
    printf("varptr is located at address: %p\n", <...>);
}
```

Exercise 2

Complete the following statements using pointers and pointer arithmetic

```
void change_array(int *arr);

int main() {
    int arr[3] = {100, 101, 102};

    // Implement and call the change_array function to set the second element of
    // arr to 123. Implement the function as pass-by-reference and do not use
    // indexing (i.e., you cannot access the array as arr[i]).
    <...>

    // Print all elements of the array using pointer arithmetic
    // (i.e., you cannot access the array as arr[i]).
    <...>
}
```

Exercise 3

Create an Arduino class in C++ that provides the following members:

Private

➡ a integer variable for storing the serial number of the Arduino card;

➡ a method for setting the serial number of the Arduino card;

Public

➡ a integer variable for storing the model of the Arduino card;

➡ a method for printing the model and the serial number of the Arduino card;

➡ a static integer variable for storing the number of instantiated Arduino card objects;

➡ a parameterized constructor to set the model and the serial number of the Arduino card and increment the number of instantiated cards;

➡ a destructor;

Exercise 3

Create a second class, named Nano, that inherits from Arduino and provides the following members:

Public

- ➡ a integer variable for storing the version of the BLE protocol;
- ➡ a method for transmitting integer data through BLE (only prints the value to be transmitted);
- ➡ a parameterized constructor to set the version of the BLE protocol and calls the constructor of the parent class Arduino, by providing a model and a serial number;
- ➡ a destructor;

Exercise 3

Main function

- ➡ declare an object of class Arduino, by providing a card model and a version number;
- ➡ print the number of instantiated cards;
- ➡ declare an object of class Nano, by providing a card model, a version number and a BLE version number;
- ➡ print the number of instantiated cards;
- ➡ print the BLE version number of the nano card;
- ➡ transmit a integer number through BLE;
- ➡ declare a pointer to class Nano;
- ➡ assign the address of the declared nano object to the pointer;
- ➡ print the BLE version number of the nano card using the pointer;
- ➡ transmit a integer number through BLE using the pointer;