Part 2 - The C Language

INF107

Florian Brandner

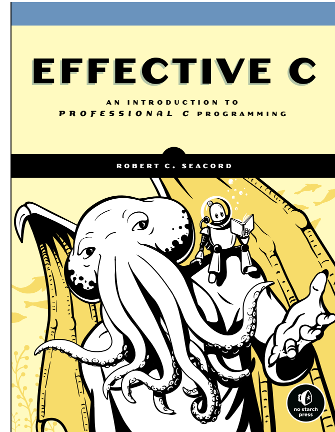
2023

Lecture 1

Welcome

Resources

- **Book:**
  EFFECTIVE C
  An Introduction to Professional C Programming
  Robert C. Seacord
  No Starch Press, 2020
  ISBN-13: 978-1-71850-104-1
- **Code casts:**
  From the popular MOOC “Programmer en C”
  Rémi Sharrock
  https://inf107.enst.fr/code-cast
- **CPP Reference:**
  https://en.cppreference.com/w/c



TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Why C?

- C has been among the most popular languages[1] of the TIOBE index since 2001.
- Widely available on most computer platforms/operating systems.
- Simple and flexible.
- Implementation basis for many other languages.
- *Good* for teaching:
  - Exposes the computer system to the programmer.
  - Full control over the computer system.
  - **Allows to make many mistakes** – (un-) fortunately.

[1] https://www.tiobe.com/tiobe-index/

History and Milestones

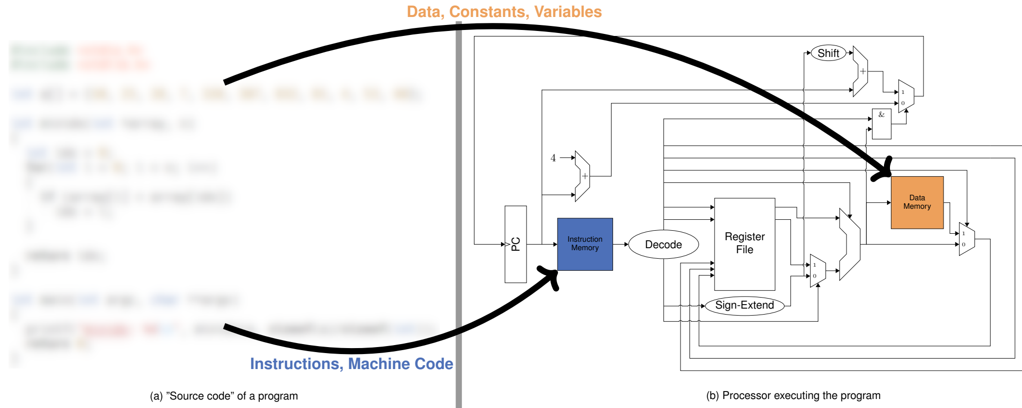
- 1972: Invented by Dennis Ritchie and Ken Thompson at Bell Telephone Laboratories
  Needed to develop their own operating system … Unix (see Part 3 of this course)
- 1989: First standard (ANSI C or C89)
  Adopted by ISO in the next year (C90)
- 1999: New ISO standard (C99) - widely supported
  Boolean type
  Integer types widht standardized sizes
- 2011: New ISO standard (C11) - well supported today
  Unicode support
  Atomics and support for multi-threading
- 2017: New ISO standard (C17) - mostly corrections
- 202x: Upcoming ISO standard (C23) currently under development

Standards

- Define what the language is (and what not).
- Standard $\neq$ Implementation
  - Not everything in standards is always implemented.
  - Some computer platforms/operating systems add extensions.
  - Some features differ between computer platforms/operating systems.
  - Some things are **implementation-defined**, **unspecified**, or even **undefined**.
- **We'll use C11 for this course**
  - Modern, still widely supported.
- For special domains
  - MISRA C: strict guidelines for safety-critical systems (automotive)
  - CERT C: for less strict guidelines for less safety-critical systems
  - Embedded C: for embedded systems with special features

Compilers - From source code to machine code

From source code to machine code (1)

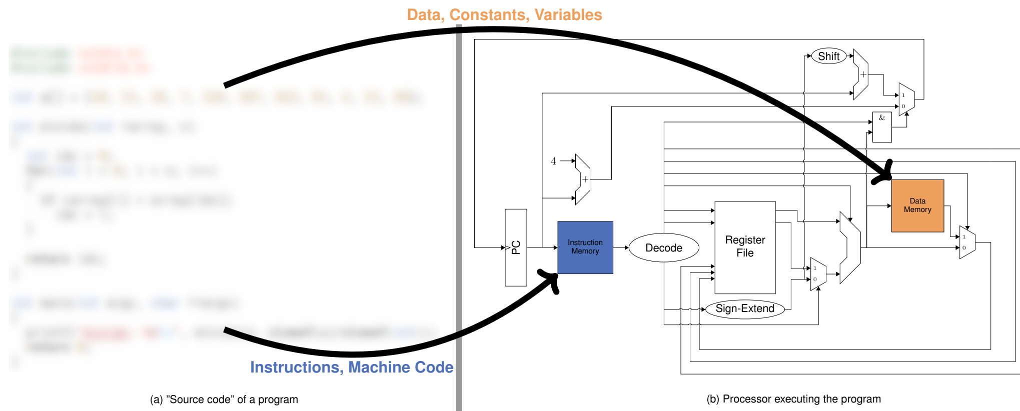


(a) "Source code" of a program

(b) Processor executing the program

The compiler translates the source code, placing machine code (instructions) and data into memory.

From source code to machine code (1)

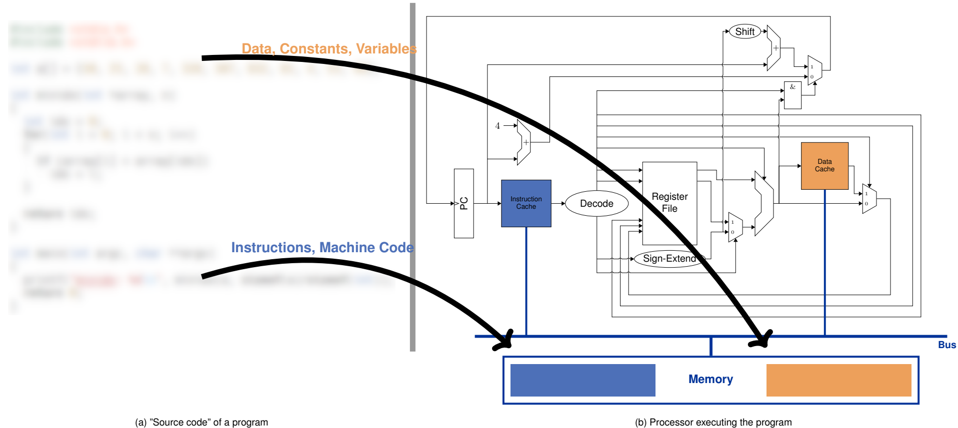


(a) "Source code" of a program

(b) Processor executing the program

In Part 1 you finished with a **Harvard Architecture**, but …

From source code to machine code (2)



(a) "Source code" of a program

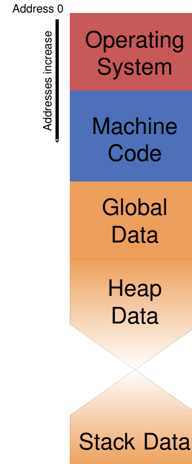
(b) Processor executing the program

… today we have a **Von Neumann Architecture** (code and data are stored in the *same* memory).

Memory Organization

We have to agree on an organization of the processor's memory:

- A part of the memory is reserved for the operating system. (code and data of the OS - see Part 3)
- Another part for the machine code of the program.
- The rest is for storing data of the program:
  - *Global data*, accessible all the time.
  - *Stack data*, accessible only temporarily.
  - *Heap data*, explicitly managed by the programmer.



Compilers

A compiler translates **high-level** source code to **low-level** binary code:

- Statements and expressions are translated to assembly or machine code.
  - Each instruction is stored at a unique address.
  - Related instructions are grouped together in close proximity (close addresses).
  - **Example:** an addition (+) becomes an `add` for a RISC-V.
- Data structures and variables are stored in memory.
  - Using a binary representation (two's-complement, BCD coding, …)
  - Each data item has a unique address.
  - Related data items are grouped together.
    - *Stack*, *heap*, or *global*
- The compiler respects the memory layout from before (i.e., code and data are disjoint)

The C Language

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Keywords

break	extern	static	auto	_Atomic (C11)
case	float	struct	goto	_Complex (C99)
char	for	switch	inline (C99)	_Generic (C11)
const	if	typedef	register	_Imaginary (C99)
continue	int	union	restrict (C99)	_Noreturn (C11)
default	long	unsigned	volatile	_Thread_local (C11)
do	return	void		_Alignas (C11)
double	short	while		
else	signed	_Alignof (C11)[2]		
enum	sizeof	_Bool (C99)[3]		
		_Static_assert (C11)		

https://en.cppreference.com/w/c/keyword

[2] Typically used through an alias: alignof

[3] Typically used through an alias: bool

A first C program (1)

- It contains comments.
  (`//` line and `/* ... */` multi-line comments)
- It includes some parts of the standard library.
  (stdio.h = Input/Output, stdlib.h = other stuff)
- It **declares** a global variable `message`.
  - The initial value of the variable is the string `"Hello World"`.
  - Its type is `const char*`.
    (we'll get back to types in a minute)
- It **defines** a function `main`
  - The `main` function has a special meaning: when executed, the program starts here.
  - Which calls the `printf` function from the IO library.
  - Returns zero.
    (and thus ends the program)

```c
/* Include functionality from the
   standard library */
#include <stdio.h>
#include <stdlib.h>

// Declare a global variable
const char message[] = "Hello World";

// Define a function
int main(int argc, char *argv[])
{
  printf("%s\n", message);
  return EXIT_SUCCESS;
}
```

Content of `hello-world.c`.

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

A First C Program (2)

To run the program we have to compile it first, only then we can execute it:

```
tp-5b07-26:~/tmp> ls
hello-world.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g hello-world.c -o hello-world

tp-5b07-26:~/tmp> ls
hello-world  hello-world.c

tp-5b07-26:~/tmp> ./hello-world
Hello World
```


TELECOM Paris

What did the compiler do?

The compiler produced the file `hello-world`:

- This is an **executable file**, i.e., a program.
- It contains **machine code**.
  (e.g., equivalent to the source code of `main`)
- It contains **binary data**.
  (e.g., the string `"Hello World"`)
- The compiler assigns the code and data to addresses in the memory.
- In order to execute the program:
  1. Load the code and data from the file into memory.
     (to the addresses specified by the compiler)
  2. Tell to processor to jump to the first instruction of the program.
  3. The processor starts executing the program …

Basic C Types

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

What is a Type?

Types are a common concept in programming languages:

- A type specifies which values are admissible at a certain point in a program (e.g., as function arguments, values of a variable, operands to an operator, ...)
- Dynamic vs. static typing:
  - **Dynamic typing:** (e.g., Python, JavaScript, ...)
    The type of values is determined and checked while the program is running.
  - **Static typing:** (e.g., Java, C, C++, OCaml, Haskell, ...)
    The type of every value is known and checked at compile-time.
- C is statically typed.

Basic C Types

In C each variable needs a fixed type. Types are grouped into classes:

- **Void type:**
  A special type without values.
- **Boolean type:**
  For boolean data with only two values (`true`/`false` or `0`/`1`).
- **Integer types:**
  For characters and integer numbers (signed or unsigned).
- **Floating-point types:**
  For floating-point numbers.

Note that the C standard does not specify the data format, but most implementations actually use a binary representation (two's complement and IEEE 754).

https://en.cppreference.com/w/c/language/type

The Void Type

The C language defines a special type `void` :

- Special type with no values.
- Used to indicate that functions do not return a value.
- Can be used to indicate that functions do not take any argument.
- Can be used with pointers (covered later in the lecture).

Boolean Type

Added only by C99, thus a rather cryptic name: `_Bool`

- Examples:

Code	Description
`_Bool done = false;`	Initializes the variable done to `false`.
`_Bool isFalse = 0;`	Initializes the variable `isFalse` to `false`.
`_Bool isTrue = 1;`	Initializes the variable `isTrue` to `true`.
`_Bool isTrueToo = .5;`	Initializes the variable `isTrueToo` to `true`.

- Alias `bool`

  An alias is defined in the library, but requires the following line in the code:

```
#include <stdbool.h>
```

https://en.cppreference.com/w/c/language/arithmetic_types#Boolean_type

Recapture: Number Representation

- **Integer numbers:**

  Usually represented using a sequence of n bits (`0`/`1`).

  - **Unsigned integers** - Simple number representation with base 2: $\sum_{i=0}^{n-1} bit_i \cdot 2^i$
  - **Signed integers:** - Uses the two's-complement representation: $-(bit_{n-1} \cdot 2^{n-1}) + \sum_{i=0}^{n-2} bit_i \cdot 2^i$
  - **Least significant bit:** bit_i with $i = 0$
  - **Most significant bit:** bit_i with $i = n - 1$

- **Floating-point numbers:** Usually based on the IEEE 754 standard.[4]
  - **Sign**: A single sign bit.
  - **Exponent**: A unsigned number (e.g., 8, 11, or 15 bits).
  - **Significand**: A unsigned number (e.g., 23, 53, or 112 bits).

[4] https://en.wikipedia.org/wiki/IEEE_754

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Integer Types

C defines several integer types:

Signed	Unsigned	Guaranteed Size[5]	In Lab
`signed char`	`unsigned char`	at least 8 bits	8 bits
`short int`	`unsigned short int`	at least 16 bits	16 bits
`int`	`unsigned int`	at least 16 bits	32 bits
`long int`	`unsigned long int`	at least 32 bits	64 bits
`long long int`	`unsigned long long int`	at least 64 bits	64 bits

The number **format is not specified** though, but usually is two's complement for signed integers.

https://en.cppreference.com/w/c/language/arithmetic_types

[5] Minimal size guaranteed by C standard in bits.

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Integer Types Aliases (1)

Integer types can be written in many variants:

Signed Type	Aliases

`short int`	`short`
	`signed short`
	`signed short int`

`int`	`signed` `signed int`

`long int`	`long`
	`signed long`
	`signed long int`

`long long int`	`long long`
	`signed long long`
	`signed long long int`

TELECOM Paris

Integer Types Aliases (2)

Unsigned integer types have aliases too (but fewer):

Signed Type	Aliases
`unsigned short int`	`unsigned short`
`unsigned int`	`unsigned`
`unsigned long int`	`unsigned long`
`unsigned long long int`	`unsigned long long`

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Examples: Integer Types and Literals

```c
unsigned char c = 225;
int i = 512;
short octal = 010;
signed hex = 0x10;
unsigned ui = 5u;
long int li = 0x20000010l;
long long lli = 0x2020000010ll;
```

Initializes the variable c to 225.
Initializes `i` to 512.
Initializes `octal` to 8 (using base 8).
Initializes hex to 16 (using base 16).
Initializes `ui` to 5 (using unsigned literal suffix).
Initializes `li` to $536\,870\,928$ (long suffix and base 16).
Initializes `lli` to $137\,975\,824\,400$ (long long suffix and base 16).

https://en.cppreference.com/w/c/language/integer_constant

TELECOM Paris

Floating-Point Types and Literals

`float`	Single-precision, usually IEEE 754 format (32 bit).
`double`	Double-precision, usually IEEE 754 format (64 bit).
`long double`	Extended-precision, usually IEEE 754 format (128 bit).

- Examples:

`float f = .5;`	Initializes the variable `f` to 0.5.
`double d = 1.2e-3;`	Initializes d to 0.0012.
`long double ld = 2.0e+308;`	Initializes `ld` to $2.0e308$.
`float hex = 0x2.ap3;`	Initializes hex to 21 ($2.625 \cdot 2^3 = 21$ in decimal).

https://en.cppreference.com/w/c/language/floating_constant

TELECOM Paris

Character Types and Symbols

`char`	Equivalent either to `signed char` or `unsigned char`, usually 8-bit ASCII value.
`wchar_t`	Usually unsigned 16-bit or 32-bit value.
`char16_t`	Usually unsigned 16-bit value, representing a UTF-16 character.
`char32_t`	Usually unsigned 32-bit value, representing a UTF-32 character.

- Examples:

`char c = 'a';`	Initializes the variable c to the symbol a (97 decimal).
`char octal = '\141';`	Initializes `octal` to the symbol a (141 octal).
`char hex = '\x61';`	Initializes hex to the symbol a (61 hexadecimal).
`char16_t c16 = u'β';`	Initializes `c16` to the symbol β (UTF-16 prefix, little beta).
`char32_t c32 = U'\u0387';`	Initializes `c32` to β (UTF-32 prefix, little beta).
`wchar_t wc = L'β';`	Initializes wc to β (wide-char prefix, little beta).

https://en.cppreference.com/w/c/language/character_constant

Character Escape Sequences

Escape Sequence	Description	ASCII	Escape Sequence	Description	ASCII
`\f`	Form feed	12	`\'`	Single quote	39
`\n`	Line feed	10	`\"`	Double quote	34
`\r`	Carriage return	13	`\?`	Question mark	63
`\t`	Horizontal tab	9	`\\`	Backslash	92
`\v`	Vertical tab	11	`\a`	Audible bell	7
`\b`	Backspace	8			
`\n`	n an octal number	n	`\uh`	h 16-bit hex number	h
`\xh`	h a hex number	h	`\Uh`	h 32-bit hex number	h

https://en.cppreference.com/w/c/language/escape
https://en.wikipedia.org/wiki/List_of_Unicode_characters

String Literals

A sequence of character symbols stored as an array is a string:

```
char hello[] = "Hello World";
char uft8[] = u8"Greek beta: β";
char16_t utf16[] = u"Beta: \u0387";
char32_t utf32[] = U"German S: ß";
wchar_t wide[] = L"German S: \u1E9E";
```

Initializes the variable `hello` to the given string.
Initializes `utf8` to the string using UTF-8 encoding.
Initializes `utf16` to the string using UTF-16 encoding.
Initializes `utf32` to the string using UTF-32 encoding.
Initializes `wide` to the string using wide characters.

https://en.cppreference.com/w/c/language/string_literal

Global Declarations and Definitions

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Structure of a C Source File

A C source file consists of …

```c
// Include code from the standard library
#include <stdio.h>
#include <stdlib.h>


int counter = 0;

void stepCounter();
int getCounter();

int main(int argc, char *argv[])
{
  // some code here
}
```

Include **header** files to *import* code from libraries (we'll get back to libraries in more detail later)

Global declarations of functions, variables, and custom types, as well as function definitions.

We call such a C source file a **translation unit**.

Global Declarations

Introduce a new **identifier** in the C program:

- An identifier is a name with a specific meaning in the program
  - Identifiers are sequence of character symbols (letters, underscore, digits, …).
  - Identifiers cannot start with a digit.
  - Identifiers are case sensitive.
- Specifies what the identifier means:
  - It may refer to a variable, function, or type.
  - It may be associated with additional properties.

https://en.cppreference.com/w/c/language/identifier

Global Declarations

Global declarations consist of three parts:

```
<Storage class and Qualifiers> <Type> <Declarators> ';'
```

- Storage class and qualifiers may appear in any order:
  - **Storage class:** For this class: `static` or `extern`.
  - **Qualifier:** For this class: `const`.
- **Type:**
  Any of the basic types, covered so far, or a custom type (yet to come).
- **Declarators:**
  One or more declarators separated by a comma (`,`), such as:
  - Identifier of a variable, optionally followed by an initializer.
  - Identifier of an array with a size in brackets (`[]`), optionally followed by an initializer.
  - identifier of a function with a parameter list in braces (`()`).

https://en.cppreference.com/w/c/language/declarations

Examples: Global Variable Declarations

```c
int counter = 0;
const short constant = 27;

extern unsigned elsewhere;
static char private = 'p';
int v1, v2, v3;
char message[] = "Hello World";
static short data[100];

int initialized[3] = {0, 1, 2};
```

Declare the variable `counter` (with initializer).

Declare `constant` as `const`, i.e., its value is not supposed to change during execution.

Declare `elsewhere` with storage class `extern`.

Declare `private` with storage class `static`.

Declare v1, v2, and v3 all at once.

Declare `message` as an array of characters (size derived).

Declare `data` as an array of 100 `short` values, stored consecutively in memory.

Declare `initialized` as an array of 3 `int` values, initialized to 0, 1, and 2 respectively.

Examples: Global Function Declarations

Code	Description
`void foo(void);`	Declare the function `foo`, does not return anything and has no argument.
`void anotherFoo();`	Declare `anotherFoo` similar to previous line.
`int bar(int);`	Declare `bar`, takes and returns an `int`.
`int anotherBar(int a);`	Declare `anotherBar` similar to previous line.
`long constantArg(const char);`	Declare `constantArg`, returns a `long int` value and takes argument with qualifier `const`.
`const long constantReturn(void);`	Declare `constantReturn`, returns a value with qualifier `const` and has no arguments.
`extern char elsewhere(int, int b);`	Declare `elsewhere` with storage class `extern` and two arguments (one without name/identifier).
`static void private(int a, int b);`	Declare `private` with storage class `static`, does not return anything and takes two arguments.

TELECOM Paris

Storage Duration and Linkage

- **Storage Duration:**
  Global identifiers are accessible during the entire execution of the program.
- **Linkage:**
  Indicates the visibility of the function/variable.
  - **Internal Linkage:**
    The function/variable is visible only within the current translation unit.
  - **External Linkage:**
    The function/variable is visible also from other translation units (aka. other C source files).

https://en.cppreference.com/w/c/language/storage_duration

Storage Classes

- By default global function/variables have **external linkage**.
- Impact of specifying the storage class for a declaration:
  - Using `static`:
    Changes linkage to be **internal**.
  - Using `extern`:
    Linkage becomes **external**, the compiler simply **assumes** that the function/variable exists.
    - The compiler *does not reserve memory space* for the code/data of the functions/variable.
    - The compiler *does not assign a memory address* in the current translation unit.
    - Variables have to be redeclared without `extern` in another translation unit.
    - Functions have to be defined without `extern` in another translation unit.

Defining Functions

Function definitions consists of four parts:

```
<Storage class and Qualifier> <Type> <Declarator> '{' <Body> '}'
```

Resembles a function declaration:

- **Storage class** and **Qualifiers**:
  Same as before, i.e., `static`/`extern` or `const` respectively.
- **Type**:
  Type of the value returned by the function.
- **Declarator**:
  The identifier and parameters of the function (same as for declarations).
- **Body:**
  The code of the function enclosed in braces.

https://en.cppreference.com/w/c/language/function_definition

Function Body

The function body consists of a sequence of **statements** and/or **declarations**:

- `if` or `if` -`else` statement.
- `switch` statement.
- `while` or `do-while` loop.
- `for` loop.
- `return` statement.
- An expressions can also be a statement (e.g., `3 + 4;`).
- **Compound statement:**
  Sequence of statements enclosed in curly braces (`{` and `}`).

https://en.cppreference.com/w/c/language/functions
https://en.cppreference.com/w/c/language/statements

Compound Statements and Scopes

Identifiers introduced by declarations are visible depending on their **scope**:

- **File scope:**
  The scope of the translation unit for global functions/variables.
- **Function scope:**
  Every function defines a new scope.
- **Block scope:**
  Every compound statement (`{` and `}`) defines a new scope.
- Scopes are nested:
  - The function scope contains the file scope.
  - A block scope contains its surrounding function or block scope.
  - …

https://en.cppreference.com/w/c/language/scope

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Declarations within Functions

All kinds of declarations are allowed within functions:

- The scope of these declarations is the currently open scope (either the function scope or the last opened block scope)
- Identifiers are only visible within the current scope or its nested scopes.
- Identifiers in nested scopes may hide identifiers from surrounding scopes.

Storage Duration and Linkage (revised)

- **Storage Duration:**
  Defines the lifetime during which a function/variable can be used:
  - **Static duration:**
    Identifiers are accessible during the entire execution of the program.
  - **Automatic duration:**
    Identifiers are accessible only when the enclosing scope is executed.
- **Linkage:**
  Indicates the visibility of the function/variable.
  - **No Linkage:**
    The variable is visible only in its enclosing scope.
  - **Internal Linkage:**
    The function/variable is visible only within the current translation unit.
  - **External Linkage:**
    The function/variable is visible also from other translation units (aka. other C source files).

https://en.cppreference.com/w/c/language/storage_duration

Storage Classes (revised)

- By default global function/variables have external linkage and static storage duration.
- **By default local variables have no linkage and automatic storage duration.**
- Impact of specifying the storage class for a declaration:
  - Using `static`:
    - Changes linkage to be **internal** for global functions/variables.
    - Changes storage duration to be **static** for local variables.
  - Using `extern`:

    Linkage becomes **external**, the compiler simply **assumes** that the function/variable exists.
    - The compiler *does not reserve memory space* for the code/data of the functions/variable.
    - The compiler *does not assign a memory address* in the current translation unit.
    - Variables have to be redeclared without `extern` in another translation unit.
    - Functions have to be defined without `extern` in another translation unit.

Example: Declarations and Scopes

```c
// File scope: message
const char message[] = "Hello World";

// File scope: message and main
int main(int argc, char *argv[])
{
  // Function scope: argc, argv, and data
  int data = 0;
  {
    // Block scope: message (hides message from file scope)
    static const char message[] = "Me First";
    printf("%s\n", message);
  }
  printf("%s\n", message);
  return EXIT_SUCCESS;
}
```

Check Yourself!

```c

const char message[] = "Hello World";

int main(int argc, char *argv[])
{
  int data = 0;
  {
    static const char message[] = "Me First";
    printf("%s\n", message);
  }
  printf("%s\n", message);
  return EXIT_SUCCESS;
}
```

1. What is the linkage/storage duration of the variable `message` from line 2?
2. What is the linkage/storage duration of the variable `message` from line 8?
3. What is the linkage/storage duration of the variable `data` from line 6?
4. What is the output of compiling this source code and running the resulting executable file?

TELECOM Paris

Answers

1. The first `message` variable is defined at file scope, with **external** linkage and **static** storage duration.
2. The second `message` variable is defined at block scope, with **no** linkage and **static** storage duration.
3. The `data` variable is defined at function scope. It has **no** linkage and **automatic** storage duration.
4. The output of compiling and running the code is:

```
tp-5b07-26:~/tmp> ls
hello-world.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g hello-world.c -o hello-world

tp-5b07-26:~/tmp> ls
hello-world  hello-world.c

tp-5b07-26:~/tmp> ./hello-world
Me First
Hello World
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Expressions (Quick)

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Expressions

Compute a single value from:

- Constants
  Same notations as seen before when we introduced types.
- Variable values
  Referenced by the variable's identifier.
- Operators
  Respecting **precedence** and **associativity**.
- Values returned by a function
  The function is **called** (or **invoked**) and returns a value.
- **Example:** `3 + 4 * a`

https://en.cppreference.com/w/c/language/expressions
https://en.cppreference.com/w/c/language/operator_precedence

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Operators and Precedence (1)

Precedence	Operator	Description	Associativity
1	++ --	Postfix increment/decrement	Left
	[ ]	Array subscripting	Left
	( )	Function call	Left
	++ --	Prefix increment/decrement	Right
2	+ -	Unary plus/minus	Right
	!	Logical NOT	Right
	~	Bitwise NOT	Right
3	* / %	Multiplication	Left
		Division	Left
		Remainder	Left
4	+ -	Addition	Left
		Subtraction	Left
5	<< >>	Bitwise shift left	Left
		Bitwise shift right	Left

Operators and Precedence (2)

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>6</td><td>< <= > >=</td><td>Less-than
Less-than-equal
Greater-than
Greater-than-equal</td><td>Left</td></tr>
<tr><td>7</td><td>== !=</td><td>Compare for equality
Compare not equal</td><td>Left</td></tr>
<tr><td>8</td><td>&</td><td>Logical AND</td><td>Left</td></tr>
<tr><td>9</td><td>^</td><td>Logical XOR</td><td>Left</td></tr>
<tr><td>10</td><td>|</td><td>Logical OR</td><td>Left</td></tr>
<tr><td>11</td><td>&&</td><td>Short-circuit AND</td><td>Left</td></tr>
<tr><td>12</td><td>||</td><td>Short-circuit OR</td><td>Left</td></tr>
<tr><td>13</td><td>? :</td><td>Conditional Operator</td><td>Right</td></tr>
<tr><td>14</td><td>=</td><td>Assignment Operator</td><td>Right</td></tr>
</table>

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Operator Precedence and Associativity

Important to understand what an expression does and how to read it:

- **Associativity:**

  Defines how expressions are braced for operators with **same** precedence.

  - **Left Associative:**

    `a - b + c + d` is equal to `((a - b) + c) + d`.

  - **Right Associative:**

    `- ~ -a` is equal to `(- (~ (- a)))`.

- **Precedence:**

  Defines how expressions are braced for operators with **different** precedence.

  `-a + b * c` is equal to `(-a) + (b * c)`.

https://en.wikipedia.org/wiki/Operator_associativity

Operator Semantics

Semantics indicates what an operator does:

- Most operators have obvious semantics …
  - Unary minus (-a) negates a number.
  - Binary plus (a + b) computes the sum of two numbers.
  - Binary multiplication (a * b) computes the product of two numbers.
  - …
- We won't explain each operator in detail, but you can consult the documentation:
  https://en.cppreference.com/w/c/language/operator_arithmetic
  https://en.cppreference.com/w/c/language/operator_logical
  https://en.cppreference.com/w/c/language/operator_comparison
  https://en.cppreference.com/w/c/language/operator_assignment

TELECOM Paris

Check Yourself!

Rewrite the following expressions with the correct bracing:

1. `a + b + c`
2. `!a * b + c`
3. `a + ++b + c`
4. `!a++ << ++b + c`

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Answers

1. `a + b + c` is the same as `(a + b) + c`.
2. `!a * b + c` is the same as `((!a) * b) + c`.
3. `a + ++b + c` corresponds to `(a + (++b)) + c`.
4. `!a++ << ++b + c` is equivalent to `(!(a++)) << ((++b) + c)`.

Statements

Statements: `if`

Comes in two variants:

(1) 'if' '(' <Cond> ')' <Sub-statement-true>

(2) 'if' '(' <Cond> ')' <Sub-statement-true> 'else' <Sub-statement-false>

- First evaluates the condition expression (<Cond>).
- If result is non-zero the (first) sub-statement is executed (<Sub-statement-true>).
- Otherwise:
  - For the first variant:
    Execute the statement following the `if`.
  - For the second variant:
    Execute the sub-statement (<Sub-statement-false>).
- Example:

```c
if (a + b < c) c = a + b;
else {
  c = b / 2;
}
```

https://en.cppreference.com/w/c/language/if

Statements: `switch` (1)

A `switch` statement conditionally executes a case:

```
'switch' '(' <Cond> ')' '{' <Cases> '}'
```

Two possible formats for a case:

```
(1) 'case' <Const-expr> ':' <Sub-statement>
(2) 'default' ':' <Sub-statement>
```

- Evaluates the condition (`<Cond>`).
- Execution continues with the case whose value (`<Const-expr>`) matches the result.
  - `<Const-expr>` has to be constant and is evaluated at compile-time.
  - The values of the different cases have to be unique.

https://en.cppreference.com/w/c/language/switch
https://en.cppreference.com/w/c/language/constant_expression

Statements: `switch` (2)

- If none of the case values matches:
  - Execution continues with the `default` case, if present.
  - Otherwise, execution continues with the statement following the `switch`.
  - Only a single `default` case is allowed.
- The cases are considered as a sequence of statements:
  - When the execution of the selected case finishes, execution simply continues in the next case.
  - One has to explicitly prevent this using a `break` statement.

https://en.cppreference.com/w/c/language/switch

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: switch

```
1   int counter = 0;
2   switch (cond) {
3     case 4: counter = counter + 1;
4     case 3: counter = counter + 1;
5     case 2: counter = counter + 1;
6             break;
7     case 1: break;
8     default: counter = 1000;
9   }
10  counter = counter * 2;
```

Execution depends on the value of cond (assume type int):

Value of cond	Lines executed	Final value of counter
1	1, 2, 7, 10	0
2	1, 2, 5-6, 10	2
3	1, 2, 4-6, 10	4
4	???	???
5	???	???

Statements: `while` Loop

In a `while` loop the sub-statement is executed repeatedly as long as the condition evaluates to true:

`'while' '(' <Cond> ')' <Sub-statement>`

- The condition expression (`<Cond>`) is evaluated.
  - If the result is non-zero the sub-statement is executed.
    - Subsequently the condition expression is reevaluated.
    - And so on and so forth …
  - If the result is zero the statement following the `while` is executed.
- Example:

```
while (counter > 5)
{
  counter = counter - 1;
}
```

https://en.cppreference.com/w/c/language/while

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Statements: do Loop

A do loop is a similar loop construct:

```
'do' <Sub-statement> 'while' '(' <Cond> ')'
```

- The sub-statement is executed first.
- Then the condition expression (<Cond>) is evaluated.
  - If the result is non-zero the sub-statement is executed again.
    - Subsequently the condition expression is reevaluated.
    - And so on and so forth.
  - If the result is zero the following statement is executed.
- Example:

```
do
{
  counter = counter - 1;
} while (counter > 5)
```

https://en.cppreference.com/w/c/language/do

Statements: `for` Loop (1)

Finally, `for` loops are just special `while` loops:

```
'for' '(' <Init> ';' <Cond> ';' <Iteration> ')' <Sub-statement>
```

- First evaluates the init expression (`<Init>`) once.
- Then the condition expression (`<Cond>`) is evaluated.
  - If the result is non-zero the sub-statement is executed.
    - Next the iteration expression (`<Iteration>`) is evaluated.
    - Subsequently the condition expression is reevaluated.
    - And so on and so forth …
  - If the result is zero the following statement is executed.

https://en.cppreference.com/w/c/language/for

Statements: `for` Loop (2)

Finally, `for` loops are just special `while` loops:

```
'for' '(' <Init> ';' <Cond> ';' <Iteration> ')' <Sub-statement>
```

… is (more or less) equivalent to the following `while` loop:

```
<Init> ';'
'while' '(' <Cond> ')'
'{'
   <Sub-statement>
   <Iteration> ';'
'}'
```


TELECOM Paris

Statements: Jumping in Loops

One may exit a loop or skip to the next iteration using jump statements:

- `break`:
  - A `break` statement can also be used in loops (recall its use for the `switch` statement).
  - It exits the loop, execution continues with the following statement after the loop.
- `continue`:
  - Skips the remaining statements in the loop.
  - Execution continues with the evaluation of the condition in a `while` or `do` loop.
  - Execution continues with the evaluation of the iteration expression in a `for` loop.

https://en.cppreference.com/w/c/language/break
https://en.cppreference.com/w/c/language/continue

Statements: `return`

In order to leave a function one can use the `return` statement:

- If the return type of the function is `void`:
  - It suffices to simply write `return;` without a return value.
  - Execution continues after the call to the function.
  - Reaching the end of such a function without an explicit `return` is equivalent to a `return`.
- If the return type of the function is not `void`:
  - A return value has to be supplied: `return <Expression> ;`.
  - Execution continues after the call to the function.
  - Reaching the end of such a function without an explicit `return` is **undefined** behavior.

https://en.cppreference.com/w/c/language/return

A First Algorithm: Division

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest = rest - divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Content of `division.c`.

A First Algorithm: Executing the Division

To run the program we have to compile it first and then execute it:

```
tp-5b07-26:~/tmp> ls
division.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g division.c -o division

tp-5b07-26:~/tmp> ls
division  division.c

tp-5b07-26:~/tmp> ./division
Hello World
5
```


TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

The `main` Function

- Is the first function to be executed of a program.
- Arguments:
  - `argc`: (always type `int`)
    The number of arguments provided to the program on the command line.
  - `argv`:
    Array of strings, one string for each command-line argument.
- Return Value: (always type `int`)
  Exit status of the program, `EXIT_FAILURE`/`EXIT_SUCCESS` on error/success.
- **Example**: `./division one 2` results in
  `argc`: 3
  `argv[0]`: `"./division"`
  `argv[1]`: `"one"`
  `argv[2]`: `"2"`

The Standard Library

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

The Standard Library

The standard library provides elementary functions needed to write programs:

- For instance:
  - Math library.
    https://en.cppreference.com/w/c/numeric
  - Time and date library.
    https://en.cppreference.com/w/c/chrono
  - File, input, and output library.
    https://en.cppreference.com/w/c/io
  - Strings library.
    https://en.cppreference.com/w/c/string
- A complete list of library files:
  https://en.cppreference.com/w/c/header

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Using Library Functionality

A **header file** needs to be included to use library functions.

- A header file is *just* a normal C file. By convention:
  - It only contains global declarations.
  - All variables are declared as external, i.e., always with `extern`.
  - Functions are not defined only declared (with or without `extern`).
- The compiler processes all declarations as if they were written in the C file.
- The compiler automatically finds function definitions.
  (we'll get back to this later)
- Example:
  `#include <stdio.h>` - Include declarations of file, input, and output library.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Header File

Here is an excerpt from the header file `math.h`:

```
<snip>
extern double acos (double __x);
extern double asin (double __x);
extern double atan (double __x);
extern double atan2 (double __y, double __x);
<snip>
extern float fminf (float __x, float __y);
extern double fmin (double __x, double __y);
extern long double fminl (long double __x, long double __y);
<snip>
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

IO Library: Formatted Output (1)

The `printf` function allows to display *formatted* information:

- Allows to print strings, characters, all basic types on the screen.
- And much more ...
- Here is its declaration:

```c
int printf(const char format[], ... );
```

  - It takes a string as parameter (`format`).
  - The dots (`...`) indicate that any number of additional parameters are accepted.
    - Such functions are called *variadic*, we won't cover them more than that.
    - Integer promotion is applied for these function arguments.
    - `float` values are promoted to `double` for these function arguments.
  - `format` specifies how to display the other parameter values.
- **Example:** `printf("A number: %d\n", 5)`

https://en.cppreference.com/w/c/io/fprintf

TELECOM Paris

IO Library: Formatted Output (2)

The `format` parameter is a special string:

- Regular characters are simply displayed on the screen.
- The % character has special meaning:
  - It indicates that the value of another parameter should be displayed.
  - The following characters indicate how the value should be displayed.
- A quick summary for now (more elaborate explanation next time):

%c	Displays a character symbol.
%d	Displays a signed integer value (types `_Bool`, `char`, `int`, or `short`) as decimal.
%u	Displays a unsigned integer value (unsigned `_Bool`, `char`, `int`, or `short`) as decimal.
%x	Displays an integer value (signed or unsigned `_Bool`, `char`, `int`, or `short`) as hexadecimal.
%f	Displays an floating-point number (`float` or `double`) as decimal.
%e	Displays an floating-point number (`float` or `double`) in exponent notation.
%s	Displays all the characters of a string.

Example: Formatted Output (1)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char c1 = 'a', c2 = 97;
  unsigned short s = 540;
  int i = 0xfbfb;
  float f = i * 1.133e5;
  static const char string[] = "Some string\nwith a line break.";

  printf("Character symbols: %c and %c are the same\n", c1, c2);
  printf("Characters as numbers: %d and 0x%x are the same\n", c1, c2);
  printf("Integer numbers (decimal) : %u and %d\n", s, i);
  printf("Integer numbers (hex): 0x%x and 0x%X\n", s, i);
  printf("Floating-point numbers: %f and %e\n", f, f);
  printf("String: %s\n", string);
  printf("Argument: %s\n", argv[0]);

  return EXIT_SUCCESS;
}
```

Content of `print.c`.

TELECOM Paris

Example: Formatted Output (2)

```
tp-5b07-26:~/tmp> ls
print.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g print.c -o print

tp-5b07-26:~/tmp> ls
print print.c

tp-5b07-26:~/tmp> ./print
Character symbols: a and a are the same
Characters as numbers: 97 and 0x61 are the same
Integer numbers (decimal) : 540 and 64507
Integer numbers (hex): 0x21c and 0xFBFB
Floating-point numbers: 7308643328.000000 and 7.308643e+09
String: Some string
with a line break.
Argument: ./print
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Check Yourself!

1. What is the purpose of the `break` statement in a `switch`?
2. What is the difference between a `while` and `do-while` loop?
3. Where does the execution of a C program start?
4. What is the difference between a C header file and a regular C source file?

TELECOM Paris

INSTITUT POLYTECHNIQUE DE PARIS

Answers

1. The `switch` statement allows to distinguish different cases, depending on the value of its condition expression. The cases within the `switch` are considered to be a sequence of statements. So, execution may simply continue with the next case. Unless a `break` statement is used. It exits the `switch` and continues execution at the statement following it.
2. When reaching (entering) a `do-while` loop the loop's body is executed once before the loop condition is verified. For `while` loops the loop condition is evaluated first, before potentially executing the loop's body.
3. Execution starts with the `main` function
   (almost: some code of the standard library is executed *earlier* to initialize the memory, e.g., setting up the stack and heap)
4. A header file only contains declarations with the `extern` keyword, e.g., it does not contain code of functions. Regular C files contain at least one declaration without the `extern` keyword.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Lab Exercises

Get familiar with the C language and compiler:

- Compile and run some existing code.
- Use a debugger to inspect running code.
  - Division
- Write a couple of simple programs:
  - Bit-level manipulation of integer values.
    (extract sign-bit of a signed integer)
  - Sieve of Eratosthenes[6]
    (compute the primes up to 100, print integers on screen)
  - Insertion sort
    (sort floating-point numbers in an array, print floats on screen)

[6] https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

TELECOM Paris

Lecture 2

Expressions (Elaborated)

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Implicit Conversion

Which type is used to perform the addition in the following code?

```
long long int a = 5000ll;
unsigned char b = 7;
double c = a + b;
```

This code contains several **implicit conversions**:

- **Initialization:**
  The value `7` has type `int` and is converted to `char`.
- **Integer promotion:**
  The value of b is *promoted* from `unsigned char` to `int` or `unsigned int` for the addition.
- **Usual arithmetic conversion:**
  The addition is performed as a `long long int`, the value of b is thus again converted.
- **Assignment:**
  The result of the addition does not match the type of `c`, which is thus converted to `double`.

https://en.cppreference.com/w/c/language/conversion

TELECOM Paris

Integer Conversion Rank

All integer types are *ranked* according to their precision:

- Every signed integer type has a unique rank.
- Unsigned integer types have the same rank as their signed counterparts.
- The rank of `char` is the same as its signed/unsigned counterparts.
- `_Bool` has the smallest rank.
- The rank of a type is larger than that of another type if its precision is larger.
- The usual order for signed integer types (by increasing rank):
  `_Bool`, `signed char`, `short`, `int`, `long int`, `long long int`

Integer Promotion

Values of types with a *small* rank are promoted to the type `int` or `unsigned int`:

- If `int` can represent all values of the smaller type it is promoted to `int`.
- Otherwise is is converted to `unsigned int`.
- All operations with small types are thus implicitly performed as `int`/`unsigned int`.
- Integer promotion is performed:
  - For all unary operators and shift operators.
  - As part of the *usual arithmetic conversion* for binary operators (except shifts).
  - Under certain conditions: To arguments of function calls.

Usual Arithmetic Conversion

Is performed for (most) binary operators:

- If one operand is a `long double`, the other operand is converted to `long double`.
- Otherwise, if one operand is a `double`, the other operand is converted to `double`.
- Otherwise, if one operand is a `float`, the other operand is converted to `float`.
- Otherwise, integer promotion is performed on both operands.
    - If the two types of the operands are the same, no further conversion is performed.
    - Otherwise, the operand whose type has a smaller rank is converted to the one with the larger rank.
    - Except if:
        - The operand with the smaller rank is unsigned.
        - The type with larger rank is signed and cannot represent all values of the smaller unsigned type.
        - In this case both values are converted to the unsigned counterpart of the larger type.

Integer Conversions

When converting from one type to another the value may change:

- If the destination type can represent the value, the value remains unchanged.
  (except for sign- or zero-extension)
- If the destination type cannot represent the value:
  - If the destination type is **unsigned**, the value is truncated.
    (unsigned numbers consequently perform modulo arithmetic)
  - If the destination type is **signed**, the result is **implementation-defined**.
    (on the lab machines: the value is truncated)

C Standard Peculiarities

You have seen that the C standard leaves things open:

- The number representation is not fixed by the standard and might not be two's complement.
- The size of integer/floating-point types is not fixed, only minimal guarantees are specified.
- The C standard uses the following convention:
  - **Implementation-Defined Behavior:**
    The operating system/compiler may chose what is to be done. The chosen behavior has to be documented and has to be applied consistently.
  - **Unspecified Behavior:**
    The behavior is covered by the standard, but may change from one execution to another, i.e., one cannot rely on the behavior.
  - **Undefined Behavior:**
    The behavior is *outside* of the standard or explicitly defined to be undefined. This is typically neither a bug in the standard nor an omission, but a deliberate choice by the standard committee.

Operators: Type Cast

One may also perform type conversions explicitly:

```
'(' <Type> ')' <Expr>
```

- Explicitly converts the result of the expression to the indicated type.
- Special case: `'(' 'void' ')' <Expr>`
  - Discards the computed value.
- **Example:**

```c
float f = .5;
int main(int argv, char *argv[]) {
  return (int)f;                    // Explicit cast.
}
```

https://en.cppreference.com/w/c/language/cast

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Operators: Array Subscripting

Allows to access individual elements of an array:

```
<expression> '[' <subscript> ']'
```

- The subscript has to be of integer type.
  - Array elements are indexed starting with 0.
  - For an array with size n the last element has index $n - 1$.
  - Subscripts that exceed the array size result in **undefined** behavior.
- **Example:**

```
int array[] = {1, 2, 3, 4};
int main(int argv, char *argv[]) {
  return array[2];                // Array subscript.
}
```

https://en.cppreference.com/w/c/language/operator_member_access#Subscript

Operators: Basic Arithmetic

Basic arithmetic operators are `+`, `-` , `*`, `/`, and `%`:

- These operators are mostly self explanatory … but may come with surprises:
- Overflow:
  - Unsigned operations are always performed using modulo arithmetic.
    (if the operation would exceed the type's range, simply to result modulo 2^n is computed)
  - Overflow for signed operations is **undefined**.
    (on the Lab machines: the result is truncated)
- Integer division is *truncated towards 0*.
  - Division by 0 is **undefined**.
  - Remainders thus always have the same sign as the dividend.
  - Example: $-11/3 = -3.666667$
    - The integer division in C (`-11/3`) yields `-3`.
    - The remainder in C (`-11%3`) yields `-2`.

https://en.cppreference.com/w/c/language/operator_arithmetic

TELECOM Paris

Operators: Bitwise Shift Operators

Shift operators are `<<` and `>>`:

- Move the bit pattern to the left or right by k positions
- For unsigned integers:
  Sets the k least-/most-significant bits to 0 for a left/right shift.
- For signed integers:
  - Sets the k least-significant bits to 0 for a left shift.
  - Sets the k most-significant bits to the initial most-significant bit (sign) for a right shift.
- Does not apply the usual arithmetic conversion, but only integer promotion.
  The result type is the type of the left operand, after promotion.
- The right operand (k) should not exceed the size of the type of the left operand and should not be negative, or else the result is **undefined**.

https://en.cppreference.com/w/c/language/operator_arithmetic

Operators: Comparison Operators

Comparisons (`==`, `!=`, `<`, …) compare the values of the two operands:

- The result is 1 when the relation holds, 0 otherwise.
- The unary logical NOT operator is equivalent to a test for zero:
  `!a` is equivalent to `0 == a`.

https://en.cppreference.com/w/c/language/operator_comparison
https://en.cppreference.com/w/c/language/operator_logical

Operators: Assignment

Be careful to not confuse the assignment operator (=) and comparison operator (==)!

```
<modifieable lvalue> = <expression>
```

- Assigns the result of the expression from the right side to the *lvalue* on the left side.
- So far we have only seen two kinds of lvalues: variables and array subscripts.
- The types of the left and right side must either be convertible or compatible:
  - Implicit conversion may occur.
  - The behavior of the implicit conversion may be implementation-defined.
- An assignment may appear in an expression and evaluates to the value of the right side: `a + (b = 2)` the assignment evaluates to `2` and thus the expression to `a + 2`

https://en.cppreference.com/w/c/language/operator_assignment

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Operators: Assignment Variants

The C language provides **compound assignment** operators:

- These operators are:
  `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
- They are just a shortcut:
  `a *= b` is equivalent to `a = a * b`

https://en.cppreference.com/w/c/language/operator_assignment

Operators: Increment and Decrement

C defines special operators to increment/decrement an lvalue:

(1) `a++` (2) `a--` (3) `++a` (4) `--a`

The above examples increment/decrement the value of `a` by 1 respectively.

- The first two (1) and (2) are called **post**-increment/-decrement operators:
  - The value of the entire expression is the initial value of `a`.
  - `a` is incremented/decremented independently.
- The next two (3) and (4) are called **pre**-increment/-decrement operators:
  - The value of the entire expression is the new (incremented) value of `a`.
  - `a` is also incremented/decremented.
  - They are the equivalent to `a += 1` and `a -= 1` respectively.

https://en.cppreference.com/w/c/language/operator_incdec

Operators: Function Calls

A function call consists of two elements:

```
<Expression> '(' <Argument list> ')'
```

- **Expression:**
  For this class: the expression may only be the identifier of the function to be called.
- **Argument list:**
  A possibly empty list of expressions separated by a comma (`,`).
- Calling a function may provoke **side-effects**:
  - Values of variables may change (before vs. after the call).
  - More generally: memory content may change.
  - The function may perform input/output operations.
  - ...

https://en.cppreference.com/w/c/language/operator_other

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Evaluation Order

The evaluation order of the terms in expressions is generally **unspecified**:

- A compiler may chose any order respecting operator precedence and associativity.
  - The order may even change for the same expression.
  - This is important for operations with side-effects:
    - Function calls.
    - Increment/decrement operators.
    - Assignment operators in expressions.
- The standard defines:
  - **Value computation:**
    Determining the value of an expression.
  - **Side-effects:**
    Modification of a memory location (e.g., variable), input/output, …

https://en.cppreference.com/w/c/language/eval_order

Example: Evaluation Order

How is the following expression evaluated?

```
f1() + f2() * f3()
```

- **Associativity and precedence:**

  ```
  f1() + (f2() * f3())
  ```
- **Value computation:**
  - Return value of `f2()` and `f3()` needed for multiplication.
  - Return value of `f1()` and the multiplication result needed for addition.
- **Side-effects:**
  - Order is not specified.
  - The call to `f3()` may be evaluated before/after `f1()` and/or `f2()`.
  - Only guarantee: side-effects of different functions are not interleaved.

Example: Evaluation Order and Undefined Behavior

The behavior of the following code snippets is **undefined**:

(1) `i++ + i`   (2) `f(i *= 5, i++)`

- The evaluation order is unspecified (also for function arguments).
- Side-effect of incrementing `i` (`i++`) in (1):
  - Has no order relative to the value computation of `i` as the second operand.
  - The value of the expression is thus **undefined**.
- Side-effects on `i` in (2):
  - Both arguments of the function call modify `i`.
  - There is no ordering of these two side-effects.
  - The outcome is thus **undefined**.

TELECOM Paris

Operators: Short-Circuit Operators

Operators with a well-defined evaluation order are `&&` and `||`:

- Perform logical AND/OR on booleans respectively.
- The value computation and side-effects of the **left operand are evaluated first**.
- The right operand is only evaluated when:
  - The left operand evaluated to a non-zero value for `&&`.
  - The left operand evaluated to zero for `||`.
  - Otherwise neither value computation nor side-effects are evaluated for the right operand.
- Example: `0 && f1()`
  Never calls `f1`, so its side-effects are never evaluated too.

https://en.cppreference.com/w/c/language/operator_logical

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Operators: Conditional Operator

The evaluation order of the conditional operator is also defined:

```
<Cond> '?' <Expr-true> ':' <Expr-false>
```

- First value computation and side-effects for expression `<Cond>` are performed.
- Depending on the result:
  - If the result is 0, the right (`<Expr-false>`) operand is evaluated.
  - Otherwise, the left (`<Expr-true>`) operand is evaluated.
  - The respective other operand is not evaluated.

https://en.cppreference.com/w/c/language/operator_other

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Check Yourself!

1. What is the difference between (1) `i++` and (2) `++i`?
2. Give an example with **undefined** behavior.
   Should your code make use of such constructs?
3. Give an example of **implementation-defined** behavior.
   Should your code make use of such constructs?
4. Give an example of **unspecified** behavior.
   Should your code make use of such constructs?
5. What is the difference between `f1() && f2()` and `f1() & f2()`?

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Answers (1)

1. Both operators increment the variable `i`, the difference is the result: (1) the expression evaluates to the initial value of `i`; (2) the result is the new (incremented) value of `i`.
2. `i << -3` is **undefined** due to the negative shift amount. You should avoid any constructs with undefined behavior in your code.
3. `signed char i = 512;` contains an implicit conversion from the constant's type `int` to `signed char`. On the lab machines the possible range of `signed char` is exceeded, the result is implementation-defined. It is sometimes difficult to avoid implementation-defined behavior, but you should avoid implementation specific code if possible.

Answers (2)

4. For the expression `f1() + f2()` the evaluation order, notably concerning side-effects, is unspecified. You should be aware of such behavior and carefully write your code to obtain the actual behavior that you want, e.g., you may rewrite the code unambiguously using two parts:
   (1) `int tmp = f1();` (2) `tmp + f2();`

5. The evaluation order of `f1() & f2()` is not specified. The calls to `f1`/`f2` my execute in any order. While for `f1() && f2()` the evaluation order is specified, `f1` is executed first. If its result is `true` also `f2` is executed, otherwise not.

A First Algorithm: Division

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest = rest - divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Content of `division.c`.

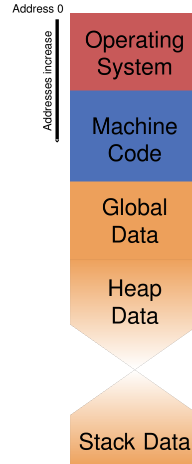
TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Memory Organization

The usual organization of the processor's memory:

- A part of the memory is reserved for the operating system.
- Another part for the machine code of the program.
- The rest is for storing data of the program:
  - *Global data*, accessible all the time.
  - *Stack data*, accessible only temporarily.
  - *Heap data*, explicitly managed by the programmer.

Where in memory did the compiler put the code and data of this first algorithm?



A First Algorithm: Memory Addresses - using nm

```
tp-5b07-26:~/tmp> ls
division  division.c

tp-5b07-26:~/tmp> nm -nS division
<snip>
400547 30 T division
400577 54 T main
<snip>
400678 0c R message
<snip>
402020 02 D data
<snip>
40203c 04 B division_result
<snip>
```

The nm tool shows all **global** variables/functions of a program:

- The first column shows the address (in hexadecimal)
- The second column the size (in hexadecimal)
- For the third column:
  - T indicates a function containing *machine code* (aka. text).
  - R indicates a variable containing *read-only data*.
  - B indicates a address containing *data that is not initialized* (block starting symbol or BSS).
  - D indicates a address containing *data that is initialized*.
- The last column shows the identifier of the variable/function.

Recall: Global variables/functions have **static** storage duration, they are loaded into memory and initialized at program start.

A First Algorithm: Memory Addresses using Debugger

Load Binary | division | reverse | NI | SI

show filesystem | fetch disassembly | jump to line | /home/brandner/work/telecom/inf10x/supports/lectures/lecture-part2-C-language/src/divisior

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  unsigned int division(unsigned int dividend, unsig
5    unsigned int result = 0;
6    for(unsigned int rest = dividend; rest >= diviso
7      rest -= divisor;
8    return result;
9  }
10
11 const char message[] = "Hello World";
12 short data = 25;
13 int division_result;
14
15 int main(int argc, char **argv) {
16   division_result = division(data, 7) + 2;
17   printf("%s\n", message);
18   printf("%d\n", division_result);
19   return EXIT_SUCCESS;
20 }
21
   (end of file)
```

threads

local variables

```
dividend 25 unsigned int
divisor 7 unsigned int
result 3 unsigned int
```

expressions

expression or variable

```
- &dividend 0x7fffffffd94c unsigned int *
  *&dividend 25 unsigned int
- &divisor 0x7fffffffd948 unsigned int *
  *&divisor 7 unsigned int
- &result 0x7fffffffd95c unsigned int *
  *&result 3 unsigned int
```

Tree

memory

breakpoints

signals

registers

```
running command: gdb

GNU gdb (GDB; SUSE Linux Enterprise 15) 11.1
Copyright (C) 2021 Free Software
```

```
gdbgui output (read-only)
Copy/Paste available in all terminals with ctrl+shift+c, ctrl+shift+v
Started new gdb process, pid 294
```

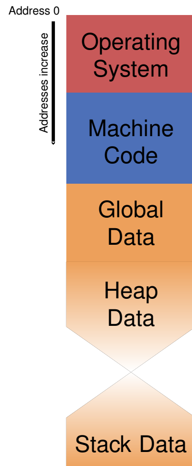
```
Program output -- Programs being debugged are connected to this terminal. You can read output and send input to the program from here.
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

A First Algorithm: Memory Addresses - Summary

What did we find:

- Machine code is stored from `0x400547` to `0x4005cb`.
- Global data is stored from `0x400678` to `0x402040`.
- Local variables are stored on the stack:
  - Stack data is stored from `0x7fffffffd94c` to `0x7fffffffd960`. (for function `division`, which was called from `main`)
  - Stack data is stored from `0x7fffffffd970` to `0x7fffffffd980`. (for function `main`)
- This matches our memory layout from before! Yay!



Automatic Storage Duration and the Stack

The stack allows the compiler to manage *temporary* data:

- The stack indicates a memory region **reserved** for local data:
  - **Stack Pointer**:
    The address where this memory region starts.
  - Decrementing/incrementing the stack pointer reserves/frees space on the stack.
- The compiler generates code for each function:
  - **Entry:**
    - Reserve new space to store temporary data.
    - The stack pointer decrements (moves down).
  - **Exit:** Free new space to store temporary data.
    - Free the temporary space again.
    - The stack pointer increments (moves up).
  - This corresponds to the *automatic storage duration* of C.

Example: The Stack during Execution (1)

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Example: The Stack during Execution (2)

Stack **before entering** the function `main`:

```
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

	64-bit Value		
Stack Address	32-bit Value	32-bit Value	Description
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv
7fffffffd9b0	00000001	00000000	Value of argc
...			
7fffffffda88	00007fffffffdf6e		Value of argv
7fffffffda90	0000000000000000		Value of argv
...			
0x7fffffffdf68	"\0\0\0\0\0\0./"		Value of argv
0x7fffffffdf70	"division"		Value of argv
...			

TELECOM Paris

Example: The Stack during Execution (3)

Stack **after entering** the function `main`:

```
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	32-bit Value	32-bit Value	Description
	64-bit Value		
7fffffffd980	00007fffffffda88		Value of argv
7fffffffd988	00000000	00000001	Value of argc
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv
7fffffffd9b0	00000000	00000001	Value of argc
	...		
7fffffffda88	00007fffffffdf6e		Value of argv
7fffffffda90	0000000000000000		Value of argv
	...		
0x7fffffffdf68	"\0\0\0\0\0\0./"		Value of argv
0x7fffffffdf70	"division"		Value of argv
	...		

Example: The Stack during Execution (4)

Stack **after entering** the function `division`:

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd958	00000007	00000019	Values of `dividend`/`divisor`
7fffffffd960	0000000000400040		Unused
7fffffffd968	00000000	00000000	Values of `result`/`rest`
7fffffffd970	00007fffffffd990		Saved register
7fffffffd978	000000000040059a		Return address (`main`)
7fffffffd980	00007fffffffda88		Value of `argv`
7fffffffd988	00000000	00000001	Value of `argc`
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of `argv`
7fffffffd9b0	00000000	00000001	Value of `argc`

...

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: The Stack during Execution (5)

Stack **before leaving** the function `division`:

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd958	00000007	00000019	Values of `dividend`/`divisor`
7fffffffd960	0000000000400040		Unused
7fffffffd968	00000004	00000003	Values of `result`/`rest`
7fffffffd970	00007fffffffd990		Saved register
7fffffffd978	000000000040059a		Return address (`main`)
7fffffffd980	00007fffffffda88		Value of `argv`
7fffffffd988	00000000	00000001	Value of `argc`
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of `argv`
7fffffffd9b0	00000000	00000001	Value of `argc`

...

TELECOM Paris
Institut Polytechnique de Paris

Example: The Stack during Execution (6)

Stack **after leaving** the function `division`:

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd980	00007fffffffda88		Value of argv
7fffffffd988	00000000	00000001	Value of argc
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv
7fffffffd9b0	00000000	00000001	Value of argc
...			
7fffffffda88	00007fffffffdf6e		Value of argv
7fffffffda90	0000000000000000		Value of argv
...			
0x7fffffffdf68	"\0\0\0\0\0\0./"		Value of argv
0x7fffffffdf70	"division"		Value of argv
...			

Compilation, Addresses, and Alignment

The compiler assigns addresses:

- In the order of declarations in the code:
  - Code/functions to fixed *global* addresses.
  - Global declarations to fixed *global* addresses.
    (Static storage duration)
  - Local declarations to relative addresses on the stack.
    (Automatic storage duration)
  - The address does not change during an object's lifetime.
- Each object has a known **size**.
- Respecting **alignment**:
  - Each type has a minimum alignment n.
  - The addresses of any object (variable) has to be a multiple of n.

https://en.cppreference.com/w/c/language/object#Alignment

Operators: `sizeof` and `_Alignof`

C provides operators to determine size and alignment:

- **Alignment:** `'_Alignof' '(' <Type> ')'`
  Works only with types, returns the minimum alignment.
  - Alternative `alignof` :
    Just an alternative name, requires `#include <stdalign.h>`.
- **Size:** `'sizeof' '(' <Type> ')'` or `'sizeof' <Expr>`
  Works with expressions and types.
- The type of both operators is `size_t`.

https://en.cppreference.com/w/c/language/sizeof
https://en.cppreference.com/w/c/language/_Alignof
https://en.cppreference.com/w/c/types/size_t

Example: Size and Alignment (1)

```c
#include <stdalign.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("char:\t\talignment:%zd size:%zd\n", _Alignof(char), sizeof(char));
  printf("short:\t\talignment:%zd size:%zd\n", _Alignof(short), sizeof(short));
  printf("int:\t\talignment:%zd size:%zd\n", _Alignof(int), sizeof(int));
  printf("long:\t\talignment:%zd size:%zd\n", _Alignof(long), sizeof(long));
  printf("long long:\talignment:%zd size:%zd\n", _Alignof(long long), sizeof(long long));
  printf("float:\t\talignment:%zd size:%zd\n", _Alignof(float), sizeof(float));
  printf("double:\t\talignment:%zd size:%zd\n", _Alignof(double), sizeof(double));
  printf("long double:\talignment:%zd size:%zd\n", _Alignof(long double), sizeof(long double));

  return EXIT_SUCCESS;
}
```

Content of `size-alignment.c`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Size and Alignment (2)

On lab machines (Linux, x86-64) the previous code yields:

```
tp-5b07-26:~/tmp> ls
size-alignment.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g size-alignment.c -o size-alignment

tp-5b07-26:~/tmp> ls
size-alignment size-alignment.c

tp-5b07-26:~/tmp> size-alignment
char:           alignment:1 size:1
short:          alignment:2 size:2
int:            alignment:4 size:4
long:           alignment:8 size:8
long long:      alignment:8 size:8
float:          alignment:4 size:4
double:         alignment:8 size:8
long double:    alignment:16 size:16
```

Derived Types

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Derived Types

In addition to the basic types, C allows to declare custom types:

- **Pointer types:**
  Represent the **address** of another object.
- `enum` **types:**
  Integer types, used to assign values to symbolic names.
- **Structure types:** (aka. *record types*)
  Regroup multiple data items under a single type.
- **Union types:**
  Allow to store values of different types at the same address (not handled here).
- **Type definitions:**
  Allow to introduce an alias for a type name.

TELECOM Paris

Derived Types: Pointers

A pointer contains the address of another object:

- Specified as part of a declarator:
  Simply add a * before the identifier.
- The pointer type indicates the type of the object.
- Address format:
  - Usually represented as unsigned integers.
  - Size may or may not match one of the integer types.
  - Format may depend on the object's type.
  - Lab machines: 64-bit unsigned integers.
- **Example:**

```c
int *pointerToInt;                      // A pointer to an integer object.
const char *message = "Hello World";    // A pointer to a string.
extern float *floatData(int index);     // Function returning a pointer.
int *anotherPointer = pointerToInt;     // A pointer initialized from another pointer.
```

https://en.cppreference.com/w/c/language/pointer

Operators: Dereferencing (*)

The object of a pointer can be accessed by *dereferencing*:

`'*' <Pointer-Expression>`

- Evaluate `<Pointer-Expression>`, i.e., compute an address.
- Then accesses the memory at that address (for reading or writing).
- **Example:**

```
int a, *b, c;                      // Declare two integer variables, a pointer, ...
extern float *floatData(int index); // ... and a function.
<snip>
int sum = a + *b;                  // Read memory from address stored in b.
*b = c;                            // Write memory at address stored in b.
*floatData(5) = *b;                // Write/read memory using pointers.
```

https://en.cppreference.com/w/c/language/operator_member_access#Dereference

Operators: Address Of (&)

One can obtain the address of any object using the unary & operator:

`'&' <Expression>`

- Works with `<Expression>` being:
  - A Variable (and also functions).
  - An array access.
  - A dereferenced pointer.
- The expression's type is a pointer type, pointing to the object's type:

```
int a;
const char message[] = "Hello World";    // Declare some variables/functions.
extern float *floatData(int index);
<snip>
int *pointerToInt = &a;                  // Initialize the pointer with the address of a.
const char* pointerToChar = &message[2]; // Initialize with address of 3rd element.
int *pointerToInt2 = &*floatData(5);     // Initialize from dereferencing a pointer.
```

https://en.cppreference.com/w/c/language/operator_member_access#Address_of

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

NULL Pointers (1)

It is sometimes useful to indicate that a pointer holds **no valid address**:

- This is known as a null pointer, written as NULL

```
int *pointerToInt = NULL;
const char *message = NULL;
```

- NULL is defined in several headers of the standard library:
  - `#include <stddef.h>` works, for instance.
  - `#include <stdlib.h>` works too.
- Caution:
  - The actual value of NULL is implementation-defined.
  - Pointers are not automatically initialized to NULL.

```
int *pointerToInt;
int a = *pointerToInt;        // May or may not be NULL.
```

https://en.cppreference.com/w/c/types/NULL

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

NULL Pointers (2)

It is possible to test for null pointers:

- Either by explicitly comparing with NULL:

```c
// Is the same as on the right.
int *pointerToInt;
<snip>
if (pointerToInt == NULL) {
  <snip>
}
```

```c
// Is the same as on the left.
int *pointerToInt;
<snip>
if (!pointerToInt) {
  <snip>
}
```

- Or using the logical/conditional operators:

```c
// Default value instead of dereferencing NULL.
int *pointerToInt;
<snip>
int a = pointerToInt ? *pointerToInt : 5;
```

```c
// False instead of dereferencing NULL.
int *pointerToInt;
<snip>
bool b = pointerToInt && *pointerToInt == 5;
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Pointers and Casts

Pointers can be converted to/from other types:

- Pointer types can be converted from/to another pointer type:
  - If the alignment is respected, otherwise the result is **undefined**.
  - Including `void*`, a generic pointer.
  - Including `NULL`.
- Pointer types can be converted from/to integer types:
  - Sometimes useful for systems programming …
  - Sometimes happens due to implicit conversion.
  - This may be very dangerous.
- **Example:**

```
extern void *newPointer();
int *pointerToInt = (int*)newPointer();          // Cast from void*.
float *pointerToFloat = (float*)newPointer();    // Cast from void*.
```

https://en.cppreference.com/w/c/language/cast

Arrays and Pointers

Arrays and pointers in C are closely related:

- An array can be seen as a special object, consisting of:
  - The actual data, i.e., the array elements placed sequentially in memory.
  - A pointer to the data.
- Pointers can be used where arrays are expected, and vice versa. But, ...
  - Not all array operations are allowed on pointers though.
    (assigning an array initializer)
  - Not all pointer operations are allowed on arrays though.
    (for example: assigning a new address)

Example: Arrays and Pointers (1)

```
#include <stdio.h>
#include <stdlib.h>

int expectArray(int data[5]) {
  return data[3];
}

int data[] = {1, 2, 3, 4, 5};
int *pointer = &data[0];

int main(int argc, char *argv[]) {
  expectArray(data);                  // Pass an array as argument (as expected).
  expectArray(pointer);               // Pass a pointer as an argument.

  printf("%d\n", data[2]);            // Array subscript on an array.
  printf("%d\n", pointer[2]);         // Array subscript on pointer

  return EXIT_SUCCESS;
}
```

Example: Arrays and Pointers (2)

```c
#include <stdio.h>
#include <stdlib.h>

int expectPointer(int *data) {
  return data[3];
}

int data[] = {1, 2, 3, 4, 5};
int *pointer = &data[0];

int main(int argc, char **argv) {       // Main can also be declared this way.

  expectPointer(data);                  // Pass an array as argument.
  expectPointer(pointer);               // Pass a pointer as an argument (as expected).

  return EXIT_SUCCESS;
}
```

Arrays as Pointers: Differences (1)

```
#include <stdio.h>
#include <stdlib.h>

const char messageArray[] = "Hello World";          // These are ...
const char *messagePointer = "Hello World";         // ... essentially the same.

const short *pointer = {1, 2, 3, 4, 5};            // Does not work.

int main(int argc, char *argv[])
{
  const char localArray[] = "Hello World";          // These are ...
  const char *localPointer = "Hello World";         // ... essentially the same.

  messagePointer = messageArray;                    // Works.
  messageArray = messagePointer;                    // Does not work.

  localPointer = localArray;                        // Works.
  localArray = localPointer;                        // Does not work.

  return EXIT_SUCCESS;
}
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Arrays as Pointers: Differences (2)

The compiler produces a couple of error messages:

```
tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g pointer2.c -o pointer2

pointer2.c:7:26: warning: initialization makes pointer from integer without a cast [-Wint-conversion]
  const short *pointer = {1, 2, 3, 4, 5};                // Does not work.
                          ^
<snip>

pointer2.c:7:38: note: (near initialization for ‘pointer’)
pointer2.c: In function ‘main’:
pointer2.c:15:16: error: assignment to expression with array type
   messageArray = messagePointer;                        // Does not work.
                ^
pointer2.c:18:14: error: assignment to expression with array type
   localArray = localPointer;                            // Does not work.
              ^
```


TELECOM Paris

Arrays and Strings

Strings are simply arrays:

- Characters placed in memory one after the other.
- The length of the string:
  - Is not stored explicitly (as in Java/Python).
  - Instead the end is indicated by a special null character (`'\0'`).
- **Example:**

```
const char messagePointer[] = "Hello World"; // <-- Implicit '\0' at end
```

results in the following memory content:

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\0'

or in decimal:

72	101	108	108	111	32	87	111	114	108	100	0

TELECOM Paris

Example: Arrays, Pointers, and Strings – argv

Stack **before entering** the function main of our division example:

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv (pointer)
7fffffffd9b0	00000001	00000000	Value of argc
	...		
7fffffffda88	00007fffffffdf6e		Value of argv[0] (string/pointer)
7fffffffda90	0000000000000000		Value of argv[1] (null pointer)
	...		
0x7fffffffdf68	"\0\0\0\0\0\0./"		Value of argv[0]
0x7fffffffdf70	"division"		Value of argv[0]
0x7fffffffdf78	"\0\0\0\0\0\0\0\0"		Value of argv[0]
	...		

TELECOM Paris

INSTITUT POLYTECHNIQUE DE PARIS

Derived Types: Structures

Allow to regroup several data items into one type:

```
'struct' <identifier> '{' <Struct-Declarators> '}'
```

- The structure consists of a sequence of declarators:
  - Each declaring a structure member.
  - Only declarations of data items are allowed (no types, no functions).
- Members are placed sequentially in memory, respecting member alignment:
  - There might be unused bytes inside a structure, called **padding**.
  - The size of the structure might be larger than the sum of the member sizes.
  - The alignment of the structure is the **maximum alignment** among members.

https://en.cppreference.com/w/c/language/struct

TELECOM Paris

Example: Structure Types (1)

```c
#include <stdalign.h>
#include <stdio.h>
#include <stdlib.h>

struct char_short_int {
  char c;          // Alignment: 1 Size: 1 + three bytes of padding.
  int i1;          // Alignment: 4 Size: 4
  short s;         // Alignment: 2 Size: 2 + two bytes of padding.
  int i2;          // Alignment: 4 Size: 4
};
struct int_short_char {
  int i1, i2;      // Alignment: 4 Size: 4+4
  short s;         // Alignment: 2 Size: 2
  char c;          // Alignment: 1 Size: 1
};

int main(int argc, char *argv[]) {
  printf("Size:%zd vs. %zd\n", sizeof(struct char_short_int), sizeof(struct int_short_char));
  printf("Align.:%zd vs %zd\n", alignof(struct char_short_int), alignof(struct int_short_char));
  return EXIT_SUCCESS;
}
```

Content of `struct.c`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Structure Types (2)

```
tp-5b07-26:~/tmp> ls
struct.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g struct.c -o struct

tp-5b07-26:~/tmp> ls
struct  struct.c

tp-5b07-26:~/tmp> ./struct
Size:16 vs. 12
Align.:4 vs 4
```

Structure Tag vs. Type Name

The identifier of a structure is called **tag**:

- The tag is not a type name.
  - Instead one has to add `struct` before the tag.
  - **Example:**

```
struct tagName { int ID; char name[20]; int size; };  // Struct with tag "tagName"
struct tagName object = {0, "Florian", 182};  // Variable with type "struct tagName".
```

https://en.cppreference.com/w/c/language/struct_initialization

- But, one may define short names (see `typedef`, coming up soon).

Operators: Structure Member Access

The members of a structure can be accessed using the `.` operator:

```
<Struct-Expression> '.' <Identifier>
```

- Allows to read/write members using their identifier

```c
struct tagName { int ID; char name[20]; int size; };
struct tagName object = {0, "Florian", 182};
struct tagName anotherObject;

anotherObject.ID = object.ID + 1; // Read/write members.
anotherObject.name[0] = 'E';
anotherObject.size = 183;
```

https://en.cppreference.com/w/c/language/operator_member_access#Member_access

Operators: Dereferencing and Member Access

C provides a shortcut to dereference and access structure members:

```
<Pointer-Expression> '->' <Identifier>
```

- Dereference the pointer expression and access a member. (works only when pointing to a structure obviously)
- **Example:**

```c
struct tagName { int ID; char name[20]; int size; };
struct tagName object = {0, "Florian", 182};
struct tagName *pointerToObject = &object;

pointerToObject->ID = 6;     // Set member after dereferencing.
pointerToObject->size++;     // Increment member after dereferencing.
```

- This is just a shortcut for:

```c
(*pointerToObject).ID = 6;     // Set member after dereferencing.
(*pointerToObject).size++;     // Increment member after dereferencing.
```

https://en.cppreference.com/w/c/language/operator_member_access%Member_access_through_pointer

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Derived Types: enum Types

Allow to regroup symbolic names and assign integer values to them:

```
'enum' <Identifier> '{' <Enumerators> '}'
```

Enumerators are separated by commas (`,`) and come in two variants:

(1) `<Identifier>`

(2) `<Identifier> '=' <Constant-Expression>`

- Each enumerator introduces a new identifier.
  - The identifier has to be unique.
  - The identifier is associated with an integer value.
  - If no value is provided:
    - The value becomes the value of the previous enumerator plus 1.
    - Or 0, if it is the first enumerator.

https://en.cppreference.com/w/c/language/enum

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: enum Types

```c
enum states {IDLE, RUNNING=2, STOPPED, DONE};

void onState(enum states s)
{
  switch (s)
  {
    case IDLE:    // IDLE is the same as 0
        // Do something.
      break;
    case RUNNING: // ... the same as 2
        // Do something.
      break;
    case STOPPED: // ... the same as 3
        // Do something.
      break;
    case DONE:    // ... the same as 4
        // Do something.
      break;
  }
}
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Derived Types: Type Definitions

C allows to introduce shortcuts for type names:

'typedef' <Type> <Identifier>

- Introduces a new identifier as an alias of the given type:
  - The two types are synonyms.
  - Values of the respective other type are admissible.
- **Examples:**

```c
struct tagName { int ID; char name[20]; int size; };
typedef struct tagName structureType;          // structureType is structure.
typedef unsigned char ageType;                 // Age is an unsigned char.

ageType age = 5;                               // Same as an integer variable.
structureType object = {0, "Florian", 182};    // A structure.
```

https://en.cppreference.com/w/c/language/typedef

Example: Simple Linked List (1)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node node_t;                  // Declare shortcut.
struct node {
  void *data;
  node_t *next;                              // Use shortcut.
};

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t node2 = {&data[2], NULL};           // Null pointer marks list end.
  node_t node1 = {(void*)&data[1], &node2};  // Explicit pointer cast int* to void*.
  node_t node0 = {&data[0], &node1};         // Explicit cast not needed for void*.

  for(node_t *tmp = &node0; tmp; tmp = tmp->next) {   // Address of operator.
    int *ptr = (int*)tmp->data;              // Dereferencing and member access + pointer cast.
    printf("%d\n", *ptr);                    // Dereferencing.
  }
  return EXIT_SUCCESS;
}
```

Content of `linked-list.c`.

Example: Simple Linked List (2)

```
tp-5b07-26:~/tmp> ls
linked-list.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g linked-list.c -o linked-list

tp-5b07-26:~/tmp> ls
linked-list  linked-list.c

tp-5b07-26:~/tmp> ./linked-list
4
55
-7
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Check Yourself!

```c
int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t node2 = {(void*)&data[2], NULL};      // Null pointer marks list end.
  node_t node1 = {(void*)&data[1], &node2};    // Pointer cast int* to void*.
  node_t node0 = {(void*)&data[0], &node1};

  for(node_t *tmp = &node0; tmp; tmp = tmp->next) {    // Address of operator.
    int *ptr = (int*)tmp->data;                // Dereferencing and member access + pointer cast.
    printf("%d\n", *ptr);                     // Dereferencing.
  }
  return EXIT_SUCCESS;
}
```

1. How does the compiler assign the address of the variable `data` (line 2)?
2. What is the alignment of the variable `node2` (line 3)?
3. What is the size of `node2` in memory?
4. Assuming that the array `data` starts at address `0x402018`, to which address points `node1.data` after the initialization of `node1`?

TELECOM Paris

Answers

1. The variable has static storage duration. It is thus stored at a global address (not on the stack). It respects the alignment of the `int` type (Lab machines: 4).
2. It is the maximum alignment among the members of the structure type `struct node`. Since it consists of two pointers, the alignment is determined by those pointers (Lab machines: 8).
3. The variable is a structure consisting of two pointers, which in most cases have the same size (though this is not guaranteed!). In that case the variable occupies twice the size of a pointer (Lab machines: $2 \cdot 8 = 16$).
4. The integers of the array `data` are placed in memory sequentially, starting with the first element. The address of the first element is thus `0x402018`. `node1.data` is initialized with the address of the second element, which has to be at `0x402018 + sizeof(int)` (Lab machines: `0x402018 + 4`, i.e., `0x40201c`).

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Lecture 3

The Standard Library

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Reporting Errors (1)

```
void perror( const char *s );
```

Most functions of the C library report errors:

- Often as a special return value.
  - The return value does not explain why the error occurred.
- Many functions in addition set the **global variable `errno`:**
  - Defined in header `errno.h`.
  - Allows to store an error code as a number.
- `perror` allows to print a *readable* error message:
  - Defined in header `stdio.h`.
  - Based on the value of `errno`.
  - First *displays* the string `s`, provided as argument, followed by `": "`, and then `errno` explained

https://en.cppreference.com/w/c/io/perror

https://en.cppreference.com/w/c/error/errno

Reporting Errors (2)

```
_Noreturn void exit( int exit_code );
```

Sometimes it does not make sens to continue after an error:

- First the error should be reported (e.g., using `perror`).
- Then the program should be terminated using the `exit` function:
  - Calling this function ends the program.
  - It takes an exit code as argument.
  - This code has the same meaning as the return value of `main`.
  - `EXIT_FAILURE` can be used to indicate an error.
- Of course it is possible to call `exit` without an error as well.
  - Use `EXIT_SUCCESS` in this case.

https://en.cppreference.com/w/c/program/exit

Managing Heap Memory

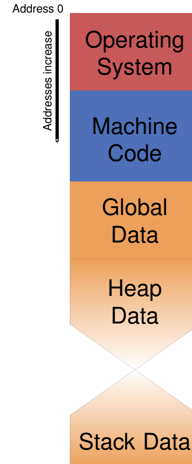
TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Memory Organization

The usual organization of the processor's memory:

- A part of the memory is reserved for the operating system.
- Another part for the machine code of the program.
- The rest is for storing data of the program:
  - *Global data*, accessible all the time.
  - *Stack data*, accessible only temporarily.
  - *Heap data*, explicitly managed by the programmer.

How can one use heap memory?



TELECOM Paris

INSTITUT POLYTECHNIQUE DE PARIS

Heap Memory

Heap memory is managed explicitly by the programmer:

- The compiler does not place data or code on the heap by itself.
- The programmer explicitly has to:
  - **Allocate:**
    Reserve an address range on the heap, depending on the size of data to be stored.
  - **Free:**
    Free an allocated address range, when the data there is not needed any longer.
- This is important so that other heap operations do not accidentally modify data there.

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Heap Memory Allocation

```
void *malloc( size_t size );
```

- Defined in the header `stdlib.h`.
- Reserves an address range of the given `size` (in bytes):
  - Returns the starting address as a pointer.
  - Returns a **null pointer**, if the heap is full (no memory space available).
- **Example**:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4);   // Allocate an array of 4 integers on the heap.
  if (!heapData) {
    perror("malloc failed");               // Print error message, if any.
    return EXIT_FAILURE;
  }
  else return EXIT_SUCCESS;                // All memory is freed implicitly.
}
```

https://en.cppreference.com/w/c/memory/malloc

TELECOM Paris

Freeing Heap Memory

```
void free( void *ptr );
```

- Defined in the header `stdlib.h`.
- Takes a valid heap pointer and frees it:
  - Valid pointer: allocated using `malloc` or a similar function and not freed before.
  - Otherwise the result is **undefined**.
- **Example**:

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4);
  if (!heapData) {
    perror("malloc failed");                    // Print error message, if any.
    return EXIT_FAILURE;
  } else {
    free(heapData);                             // Explicitly free heap data.
    return EXIT_SUCCESS;
  }
}
```

https://en.cppreference.com/w/c/memory/free

Heap Memory Reallocation

```
void *realloc( void *ptr, size_t new_size );
```

- Defined in the header `stdlib.h`.
- Takes a heap pointer and changes the size of the reserved address range:
  - The new size can be smaller or larger.
  - It may or may not return the same pointer.
  - If the pointer changes, data is copied using the old size.
- The pointer needs to be a valid heap pointer (see `free`).
- Returns a **null pointer**, if the heap is full (no memory space available).
- **Example**:

```
int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4);
  if (heapData = realloc(heapData, sizeof(int))) {  // Free some space, but not all.
    perror("realloc failed");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

https://en.cppreference.com/w/c/memory/realloc

TELECOM Paris

File Input and Output

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Manipulating Files

Files are managed by the operating system (OS):

- One has to tell the OS, which files will be manipulated and how.
- **Opening a file:**
  Tell the OS that a file is going to be manipulated.
- **Closing a file:**
  Tell the OS that the program no longer manipulates the file.
- **File operations:**
  Tell the OS that the file will be read and/or (over-)written.

The C standard library provide a higher-level API for manipulating files, based on the abstraction of (buffered) **I/O streams**.

Opening a File (1)

```
FILE *fopen( const char *filename, const char *mode );
```

- Defined in the header `stdio.h`.
- Open the given file for input/output operations.
  - `filename`:
    The name of the file to open.
  - `mode`:
    Indicates how the file will be manipulated (see next page).
- Returns a pointer to a `FILE` structure:
  - This is called a **file stream**.
  - The data structure itself is **unspecified**.
  - Programmers only manipulate the pointer.
  - On error a **null pointer** is returned.

https://en.cppreference.com/w/c/io/fopen

https://en.cppreference.com/w/c/io/FILE

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Opening a File (2)

Possible modes for `fopen`:

`mode`	Description	File exists	File does not exist
`"r"`	Reading only	Read from start	Error
`"w"`	Writing only	Overwrite file	Create file
`"a"`	Writing only	Append at end	Create file
`"r+"`	Read/write	Read/write from start	Error
`"w+"`	Read/write	Overwrite file	Create file
`"a+"`	Read/write	Read/write at end	Create file

https://en.cppreference.com/w/c/io/fopen

Closing a File

```
int fclose( FILE *stream );
```

- Defined in the header `stdio.h`.
- Closes the given file:
  - The OS makes sure that all data is written to the storage device.
  - The file can no longer be manipulated using the `FILE` pointer.
- Returns 0 on success
  - On error the constant `EOF` is returned.
    (some negative value, often -1)

https://en.cppreference.com/w/c/io/fclose

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Special File Streams

The C library defines three special file streams:

- Defined in the header `stdio.h`.
- `stdout`:
  A stream (aka. a `FILE` structure) for regular output (write only).
- `stderr`:
  A stream to report errors (write only).
- `stdin`:
  A stream to read input (read only).
- These streams are opened automatically on program start.

https://en.cppreference.com/w/c/io/std_streams

TELECOM Paris

Example: File Output (1)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  FILE *f = fopen("output.txt", "w");      // Open/create file for writing.
  if (!f) {
    perror("fopen failed");
    return EXIT_FAILURE;
  }
  else {
    fprintf(f, "Hello World %d\n", 57);   // Write some text into the file.

    if (fclose(f)) {                      // Close the file.
      perror("fclose failed");            // Report an error message (on stderr)
      return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
  }
}
```

Content of `file-output.c`.

TELECOM Paris

Example: File Output (2)

```
tp-5b07-26:~/tmp> ls
file-output.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g file-output.c -o file-output

tp-5b07-26:~/tmp> ls
file-output  file-output.c

tp-5b07-26:~/tmp> ./file-output

tp-5b07-26:~/tmp> ls
file-output  file-output.c  output.txt

tp-5b07-26:~/tmp> cat output.txt
Hello World 57
```

Formatted File Output

```
int printf( const char *restrict format, ... );
int fprintf( FILE *restrict stream, const char *restrict format, ... );
int sprintf( char *restrict buffer, const char *restrict format, ... );
```

- Same principle as plain `printf`:
  - `fprintf`:
    - Takes an additional file argument.
    - Writes into that file.
  - `printf`:
    - Actually the same as `fprintf` writing to `stdout`.
  - `sprintf`: takes an additional file argument.
    - Takes a pointer to an array as additional argument.
    - Writes into that array (buffer).
- Format strings are the same.
- **Return value:**
  - Number of characters written to file/buffer.
  - On error a **negative number** is returned.

https://en.cppreference.com/w/c/io/fprintf

Format Strings Revisited (1)

`'%' <flags>? <width>? ('.' <width>?) <size modifier>? <format>`

- `width`:
  An integer number or `'*'`
  - For strings: Indicates the Number of characters to display.
  - For numbers: Number of digits to display (before or after the `'.'`).
- `flags`:
  On ore more of the following control characters:

Control Character	Description
`'-'`	Align printed text to the left (see `width`).
`'+'`	Always display sign symbol (`+`) also for positive numbers.
`' '`	Print a space instead of sign for positive numbers.
`'0'`	Padding with `'0'` characters instead of spaces (`' '`).

Format Strings Revisited (2)

- `modifier`:
  Indicate the type of integer and floating-point numbers to print.

Control String	Description
(default)	Expect a value of type `int` when printing numbers.
`"hh"`	Expect type `signed char`/`unsigned char`.
`"h"`	Expect type `short`/`unsigned short`.
`"l"`	Expect type `long`/`unsigned long` or `wchar_t`.
`"ll"`	Expect type `long long`/`unsigned long long`.
`"z"`	Expect type `size_t`.
`"L"`	Expect type `long double`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Format Strings Revisited (3)

- `format`:

Control Character	Description
`'c'`	Displays a character symbol (use modifier).
`'d'`	Displays a signed integer value as decimal (use modifier).
`'u'`	Displays a unsigned integer value as decimal (use modifier).
`'x'`	Displays an integer value as hexadecimal (use modifier).
`'f'`	Displays an floating-point number as decimal (accepts modifier `L`).
`'e'`	Displays an floating-point number in exponent notation (accepts modifier `L`).
`'s'`	Displays all the characters of a string.
`'p'`	Displays the address of a pointer.

Example: Formatted Output (1)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  short s = 540;
  int i = 0xfbfb;
  float f = i * 1.133e5;
  static const char string[] = "|Left Aligned|";

  printf("size_t:%zd\n", sizeof(int));
  printf("Integer numbers (decimal): % hd and %+d\n", s, i);     // Printing of sign.
  printf("Integer numbers (hex): 0x%8x and 0x%08X\n", s, i);     // Padding with ' ' or '0'.
  printf("Floating-point numbers: %020.3f and %12.3e\n", f, f);  // Width before/after '.'.
  printf("String: %-20s is aligned\n", string);                  // Left aligned.

  return EXIT_SUCCESS;
}
```

Content of `print2.c`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Formatted Output (2)

```
tp-5b07-26:~/tmp> ls
print2.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g print2.c -o print2

tp-5b07-26:~/tmp> ls
print2 print2.c

tp-5b07-26:~/tmp> ./print2
size_t:4
Integer numbers (decimal):   540 and +64507
Integer numbers (hex): 0x     21c and 0x0000FBFB
Floating-point numbers: 0000007308643328.000 and    7.309e+09
String: |Left Aligned|       is aligned
```

Formatted File Input

```
int scanf( const char *restrict format, ... );
int fscanf( FILE *restrict stream, const char *restrict format, ... );
int sscanf( const char *restrict buffer, const char *restrict format, ... );
```

- Same principle as for `printf`:
  - `fscanf`:
    - Takes a file argument.
    - Reads from that file depending on format string.
  - `scanf`:
    - Actually the same as `fscanf` reading from `stdin`.
  - `sscanf`: takes an additional file argument.
    - Takes a pointer to an array as argument.
    - Reads from that array (buffer).
- Format strings are similar, but not quite the same.
- **Return value:**
  - Number of values read successfully.
  - Or EOF in case of an error.

https://en.cppreference.com/w/c/io/fscanf

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Format Strings for Input (1)

Same basic idea as for `printf`:

- Regular characters (except whitespace and `'%'`):
  - Indicate that precisely such a character needs to be read.
  - Otherwise an error is signaled.
- Whitespace characters (e.g., `' '`, `'\t'`, `'\n'`, ...):
  - Consume all available consecutive whitespaces.
  - It is sufficient to indicate a single whitespace.
- The `'%` character again has special meaning:
  - It indicates that a value should be read from the file.
  - **Always expects an address where to store the read value, i.e., a pointer or array.**

https://en.cppreference.com/w/c/io/fscanf

Format Strings for Input (2)

```
'%' '*'? <width>? <size modifier>? <format>
```

- A preceding `*` indicates that the value should not be assigned to a pointer, just read.
- `width`:
  Indicates the maximum width to read (recommended particularly for strings).
- `size modifier`:
  Same as for `printf`, i.e., (`"hh"`, `"h"`, `"l"`, `"ll"`, `"z"`, and L).
- `format`:
  Same as for `printf`, i.e., (`'c'`, `'s'`, `'d'`, `'u'`, `'x'`, `f`, and e).
  - Reading a string with `"%s"` adds the null character.
    - Arrays/pointers receiving a string thus need size `width + 1`.
  - Indicating a width with `"%c"` reads `width` characters.
    - No null character is added.
  - Reading numbers skips leading whitespaces.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Formatted Input (1)

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  FILE *f = fopen(argv[1], "r");                   // Open file for reading.
  if (!f) {
    perror("fopen failed"); return EXIT_FAILURE;
  } else {
    char c, s[50];
    int i;
    int read = fscanf(f, "%s%d%c\n", s, &i, &c);  // Read a string, an integer, and a character.
    if (read != 3) {
      perror("fscanf failed"); return EXIT_FAILURE;
    }
    printf("%s\n%d\n0x%x\n", s, i, c);
    if (fclose(f)) {                               // Close the file.
      perror("fclose failed"); return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
  }
}
```

Content of `file-input.c`.

TELECOM Paris

Example: Formatted Input (2)

```
tp-5b07-26:~/tmp> ls
file-input.c  input.txt

tp-5b07-26:~/tmp> cat input.txt
String_without_whitespace_plus_null_character
3
a

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g file-input.c -o file-input

tp-5b07-26:~/tmp> ls
file-input file-input.c input.txt

tp-5b07-26:~/tmp> ./file-input no-file.txt
fopen failed: No such file or directory

tp-5b07-26:~/tmp> ./file-input input.txt
String_without_whitespace_plus_null_character
3
0xa
```

Check Yourself!

Something (maybe) *unexpected* happened in the previous example.

1. Can you spot it?

   Hint:

   `hexdump` shows the content of `input.txt` as hexadecimal numbers (left) and characters (right).

```
tp-5b07-26:~/tmp> hexdump -C input.txt
00000000  53 74 72 69 6e 67 5f 77  69 74 68 6f 75 74 5f 77  |String_without_w|
00000010  68 69 74 65 73 70 61 63  65 5f 70 6c 75 73 5f 6e  |hitespace_plus_n|
00000020  75 6c 6c 5f 63 68 61 72  61 63 74 65 72 0a 33 0a  |ull_character.3.|
00000030  61 0a                                             |a.|
00000032
```

2. Correct the format string to get the intended behavior.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Answer

1. The character read by `scanf` is not the one that you might have expected:
   - The input file seemingly ends with a character `'a'`, right?
   - However, there are also *line feed* characters (see Escape Sequences).

```
tp-5b07-26:~/tmp> hexdump -C input.txt
00000000  53 74 72 69 6e 67 5f 77  69 74 68 6f 75 74 5f 77  |String_without_w|
00000010  68 69 74 65 73 70 61 63  65 5f 70 6c 75 73 5f 6e  |hitespace_plus_n|
00000020  75 6c 6c 5f 63 68 61 72  61 63 74 65 72 0a 33 0a  |ull_character.3.|
00000030  61 0a                                             |a.|
```

Character read!

   - The form feed character (`'\n'`, aka. `0xa`) was read, not the `'a'`.
2. It is sufficient to skip the whitespace(s) after reading the integer number, i.e., it suffices to insert a space after the `'d'`.

Example: Formatted Input Corrected (1)

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  FILE *f = fopen(argv[1], "r");                    // Open file for reading.
  if (!f) {
    perror("fopen failed"); return EXIT_FAILURE;
  } else {
    char c, s[50];
    int i;
    int read = fscanf(f, "%s%d %c\n", s, &i, &c);  // Read a string, an integer, and a character.
    if (read != 3) {
      perror("fscanf failed"); return EXIT_FAILURE;
    }
    printf("%s\n%d\n0x%x\n", s, i, c);
    if (fclose(f)) {                                // Close the file.
      perror("fclose failed"); return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
  }
}
```

Content of `file-input2.c`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Formatted Input Corrected (2)

```
tp-5b07-26:~/tmp> ls
file-input2.c  input.txt

tp-5b07-26:~/tmp> cat input.txt
String_without_whitespace_plus_null_character
3
a

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g file-input2.c -o file-input2

tp-5b07-26:~/tmp> ls
file-input2 file-input2.c input.txt

tp-5b07-26:~/tmp> ./file-input2 no-file.txt
fopen failed: No such file or directory

tp-5b07-26:~/tmp> ./file-input2 input.txt
String_without_whitespace_plus_null_character
3
0x61
```


TELECOM Paris

Strings

Strings are just arrays/pointers:

- Comparison (==) and assignment (=):
  - Do not compare/copy characters.
  - Instead **operate on pointers**.
  - May produce (maybe) confusing results:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  char sa[] = "aaa";
  char sb[] = "aaa";
  // Always prints "false".
  if (sa == sb)
    printf("true\n");
  else
    printf("false\n");
  return EXIT_SUCCESS;
}
```

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  char *sa = "aaa";
  char *sb = "aaa";
  // May print "true" or "false".
  if (sa == sb)
    printf("true\n");
  else
    printf("false\n");
  return EXIT_SUCCESS;
}
```

- Operator cannot be used to concatenate strings or numbers.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

String Operations

```
int   strncmp( const char* lhs, const char* rhs, size_t count );
char *strncpy( char *restrict dest, const char *restrict src, size_t count );
char *strncat( char *restrict dest, const char *restrict src, size_t count );
char *strndup( const char *src, size_t size );
```

- Defined in header `string.h`.
- `strncmp`: Compare strings by lexicographic order, at most `count` characters. (returns 0 **when the strings are equal**, else positive or negative number).
- `strncpy`: Copy at most `count` characters from one string to another.
- `strncat`: Concatenate at most `count` characters of one string to another.
- `strndup`: Makes a copy of the string by allocating a new string on the heap.

https://en.cppreference.com/w/c/string/byte/strncmp
https://en.cppreference.com/w/c/string/byte/strncpy
https://en.cppreference.com/w/c/string/byte/strncat
https://en.cppreference.com/w/c/string/byte/strndup

Memory Area Operations

```c
int   memcmp( const void* lhs, const void* rhs, size_t count );
void *memcpy( void *restrict dest, const void *restrict src, size_t count );
void *memset( void *dest, int c, size_t count );
```

- Defined in header `string.h`.
- `memcmp`: Compare memory areas (**“byte strings”**) by lexicographic order, looking at the first `count` bytes.
  (returns 0 **when the memory areas are equal**, else positive or negative number).
- `memcpy`: Copy precisely `count` bytes from one area to another.
- `memset`: Set precisely `count` bytes to the value of `c`.

https://en.cppreference.com/w/c/string/byte/memcmp
https://en.cppreference.com/w/c/string/byte/memcpy
https://en.cppreference.com/w/c/string/byte/memset

Manipulating Addresses and Pointers

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Manipulating Addresses and Pointers

In C memory addresses and pointers can be manipulated explicitly:

- **Casting:**
  - It is possible to cast integer values to pointers.
    (many aspects of this are **implementation-defined**)
  - It is possible to cast one pointer type to another.
    - Casts to/from `void*` (e.g., as for library functions before).
    - Casts to/from `char*` to access individual bytes.
- **Pointer Arithmetic:**
  - It is possible to address objects relative to pointers using `+`/`-` or `[]`.
  - Increment/decrement pointers.
- In all cases:
  - Alignment has to be respected.
  - Computed addresses using pointers have to be valid.
    (correspond to a global, local, or heap object)
  - Otherwise the result is usually **undefined**.

Addresses and Array Subscripting

Access the n-th element from the start of the pointer/array:

```c
#include <stdio.h>

// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  // Point to beginning of array.
  int *ptr = array;

  printf("ptr    : %p\n", ptr);
  printf("&ptr[2]: %p\n", &ptr[2]);
  printf("ptr[2] : %d\n", ptr[2]);

  return EXIT_SUCCESS;
}
```

```
tp-5b07-26:~/tmp> ./pointer-address
ptr    : 0x402020
&ptr[2]: 0x402028
ptr[2] : 3
```

- Memory layout:

Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Value of
0x402020	0x00000001	0x00000002	array
0x402028	0x00000003	0x00000004	array
0x402030	0x00000005	0x00000006	array
...	...	...	...
0x7f....	0x0000000000402020		ptr

- Memory computation:

  $\texttt{0x402020} + 2 * \texttt{sizeof(int)}$

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Addresses and Pointer Arithmetic

Actually is the same as array subscripting:

```
#include <stdio.h>

// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  // Point to beginning of array.
  int *ptr = array;

  printf("ptr        : %p\n", ptr);
  printf("ptr + 2    : %p\n", ptr + 2);
  printf("*(ptr + 2): %d\n", *(ptr + 2));

  return EXIT_SUCCESS;
}
```

```
tp-5b07-26:~/tmp> ./pointer-address2
ptr        : 0x402020
ptr + 2    : 0x402028
*(ptr + 2): 3
```

- Memory layout:

Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Value of
0x402020	0x00000001	0x00000002	array
0x402028	0x00000003	0x00000004	array
0x402030	0x00000005	0x00000006	array
...	...	...	...
0x7f....	0x0000000000402020		ptr

- Memory computation:

$\texttt{0x402020} + 2 * \texttt{sizeof(int)}$

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Pointer Arithmetic

Array subscripting and pointer arithmetic are similar:

- Compute an address relative to start of the array/pointer
  - `<Pointer-address> + <Index> * sizeof( <Type> )` or `<Pointer-address> - <Index> * sizeof( <Type> )`
  - The computed address might be smaller or larger than the pointer's address.
  - The index is automatically scaled by the element size (`sizeof`).
- Pointers can also be incremented or decremented
  - Use the pre-/post- inc-/decrement operators (`++` or `--`).
  - The address computation is the same.
  - Writes resulting address back into the pointer.

TELECOM Paris

Example: Pointer Arithmetic (1)

```c
#include <stdio.h>
#include <stdlib.h>

// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  int *ptr = array;                         // Point to beginning of array.
  for(int i = 0; i < 3; i++) {              // Increment pointer (element-wise).
    printf("ptr: %p\t*ptr: ", ptr);
    printf("0x%08x ", *ptr++); printf("0x%08x\n", *ptr++);
  }

  printf("\nbyteswise:\n");                 // Increment pointer (byte-wise).
  for(char *charptr = (char*)array; charptr < (char*)array + sizeof(array); charptr++)
    printf("%02x", *charptr);
  printf("\n");

  return EXIT_SUCCESS;
}
```

Content of `pointer-increment.c`.

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Example: Pointer Arithmetic (2)

```
tp-5b07-26:~/tmp> ls
pointer-increment.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g pointer-increment.c -o pointer-increment

tp-5b07-26:~/tmp> ls
pointer-increment pointer-increment.c

tp-5b07-26:~/tmp> ./pointer-increment
ptr: 0x402030   *ptr: 0x00000001 0x00000002
ptr: 0x402038   *ptr: 0x00000003 0x00000004
ptr: 0x402040   *ptr: 0x00000005 0x00000006

byteswise:
010000000200000003000000040000000500000006000000
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Check Yourself!

```c
// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};
int main(int argc, char *argv[]) {
  int *ptr = array;                          // Point to beginning of array.
  for(int i = 0; i < 3; i++)                 // Increment pointer (element-wise).
    printf("ptr: %p\t*ptr: 0x%08x 0x%08x\n", ptr, *ptr++, *ptr++);

  printf("\nbyteswise:\n");                  // Increment pointer (byte-wise).
  for(char *charptr = (char*)array; charptr < (char*)array + sizeof(array); charptr++)
    printf("%02x", *charptr);
  printf("\n");

  return EXIT_SUCCESS;
}
```

1. This expression `(char*)array + sizeof(array)` seems wrong, since the index is scaled by the element size of the pointer, i.e., the compiler should compute something like `sizeof(array) * sizeof(int)`. The code thus should read the memory beyond the array. Where is the error in this reasoning?
2. There is actually something problematic in this code? Can you spot it?

Answer

1. The precedence of the cast, which is stronger than the addition, the expression thus has to be read as `((char*)array) + sizeof(array)`. The reasoning from above is thus not entirely wrong. The only issue is the element size, which is not `sizeof(int)` but `sizeof(char)`, since the pointer arithmetic operates on a `char` pointer.

2. The following code line is problematic:

```
printf("ptr: %p\t*ptr: 0x%08x 0x%08x\n", ptr, *ptr++, *ptr++);
```

   - The evaluation order of the side-effects of call arguments is **unspecified**.
   - The same variable `ptr` is modified twice.
   - Consequently, the behavior of this expression is **undefined**.

TELECOM Paris

Lab Exercises

Work with pointers, structures, dynamic memory allocation, formatted input and output:

- Find a delimiter character in an array.
  - Simple pointer/array manipulation.
- Define a structure and manipulate a simple structure.
  - Simple pointer/structure manipulation.
  - Formatted output (`fprintf`, ...).
- Read formatted data into a (static) structure.
  - Formatted input and output (`fscanf`, `fprintf`, ...).
  - String manipulation (`strndup`, ...).
  - Error handling (`perror`, `exit`).
- Manipulating a structure on the heap.
  - Pointer manipulation.
  - Memory management (`malloc`, `free`, `realloc`).

Lecture 4

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

The Preprocessor

Preprocessor Directives

C programs actually consist of *two* languages:

- **Preprocessing directives:**
  - All lines that start with the symbol #.
  - **Examples:**
    - `#include <stdio.h>`
    - `#define NDEBUG`
    - `#ifndef _STDIO_H`
    - `#endif /* <stdio.h> included.  */`
- The actual C code:
  - Declarations, functions, expressions, statements.
  - Does not comprise preprocessing directives.

https://en.cppreference.com/w/c/preprocessor

The Preprocessor

A special compiler phase handling preprocessor directives:

- Performs substitutions of text/code only.
- Supports conditionals:
  - Conditional inclusion of text/code.
  - Conditionally triggering compiler errors.
- Provides a simple macro language:
  - Macros are a form of functions that may take arguments.
  - Special support for string manipulation.
  - Allow to produce text/code.
- Allows to obtain information (source file, position in file).

https://en.cppreference.com/w/c/preprocessor

Preprocessor: `#include`

Substitute the current line with the contents of a file:

- Comes in two variants:
  - `'#include' '<' <Header-name> '>'`
    Searches for a header file in *standard directories*, e.g., of the standard library.
  - `'#include' '"' <File-name> '"'`
    Searches for a header file in *user-specified directories*, e.g., of the current project.
- The included file contents is preprocessed in-turn.

https://en.cppreference.com/w/c/preprocessor/include

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: #include (1)

```c
// Declarations in header.

typedef struct node node_t;
struct node {
  void *data;
  node_t *next;
};
```

Content of `linked-list2.h`.

```c
#include <stdio.h>          // Include system header files.
#include <stdlib.h>

#include "linked-list2.h"    // Include project header file.

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t node2 = {&data[2], NULL};
  node_t node1 = {(void*)&data[1], &node2};
  node_t node0 = {&data[0], &node1};

  for(node_t *tmp = &node0; tmp; tmp = tmp->next) {
    int *ptr = (int*)tmp->data;
    printf("%d\n", *ptr);
  }
  return EXIT_SUCCESS;
}
```

Content of `linked-list2.c`.

TELECOM Paris

Example: #include (2)

We can ask gcc to only run the preprocessor using the -E option:

```
tp-5b07-26:~/tmp> ls
linked-list2.c  linked-list2.h

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g linked-list2.c -E -o linked-list2.i

tp-5b07-26:~/tmp> ls
linked-list2.c  linked-list2.h  linked-list2.i
```

The content of linked-list2.i is shown on the next slide.

TELECOM Paris

INSTITUT POLYTECHNIQUE DE PARIS

Example: #include (3)

```
745 # 1013 "/usr/include/stdlib.h" 3 4
746 # 1014 "/usr/include/stdlib.h" 2 3 4
747 # 1023 "/usr/include/stdlib.h" 3 4
748
749 # 3 "linked-list2.c" 2
750
751 # 1 "linked-list2.h" 1
752
753 # 1 "linked-list2.h"
754 typedef struct node node_t;
755 struct node {
756   void *data;
757   node_t *next;
758 };
759 # 5 "linked-list2.c" 2
```

```
760 int main(int argc, char *argv[]) {
761   static int data[] = {4, 067, 0xfffffff9};
762   node_t node2 = {&data[2],
763 # 8 "linked-list2.c" 3 4
764                          ((void *)0)
765 # 8 "linked-list2.c"
766                              };
767   node_t node1 = {(void*)&data[1], &node2};
768   node_t node0 = {&data[0], &node1};
769
770   for(node_t *tmp = &node0; tmp; tmp = tmp->next) {
771     int *ptr = (int*)tmp->data;
772     printf("%d\n", *ptr);
773   }
774   return
775 # 16 "linked-list2.c" 3 4
776         0
777 # 16 "linked-list2.c"
778                       ;
779 }
```

Content of `linked-list2.i`.

Preprocessor: `#define`

Define a macro that can later be used for text substitutions:

```
'#define' <Identifier> ('(' <Parameters>')')?
<Text>?
```

- `Identifier`

  The identifier of the macro (used to refer to the macro for substitution).
- `Parameters`

  Optional list of identifiers, separated by commas (`,`), defining macro parameters.
- `Text`

  Text that replaces the macro upon use.

https://en.cppreference.com/w/c/preprocessor/replace

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Example: #define (1)

```c
#include <stdio.h>
#include <stdlib.h>

// An empty macro.
#define EMPTY
// A macro defining a constant.
#define SIZE 10u
// A function-like macro.
#define ACCESS(x) ((x) >= (SIZE) ? 0 : array[x])
  // Tip: always parenthesize macro params (x) to avoid precedence issues

int array[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

int main(int argc, char *argv[]) {
  int idx = 0;
  while (scanf("%d", &idx) == 1) {
    // Special macros __FILE__ and __LINE__
    printf("%s: %d: %d\n", __FILE__, __LINE__, ACCESS(idx));
  }
  return 0 EMPTY;
}
```

Content of `macro.c`.

TELECOM Paris

Example: #define (2)

We can ask gcc to only run the preprocessor using the -E option:

```
tp-5b07-26:~/tmp> ls
macro.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g macro.c -E -o macro.i

tp-5b07-26:~/tmp> ls
macro.c  macro.i
```

The content of macro.i is shown on the next slide.

Example: #define (3)

```
521  # 858 "/usr/include/stdio.h" 3 4
522  extern int __uflow (FILE *);
523  extern int __overflow (FILE *, int);
524  # 873 "/usr/include/stdio.h" 3 4
525
526  # 2 "macro.c" 2
527  # 10 "macro.c"
528
529  # 10 "macro.c"
530  int array[10u] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
531
532  int main(int argc, char *argv[]) {
533    int idx = 0;
534    while (scanf("%d", &idx) == 1) {
535
536      printf("%s: %d: %d\n", "macro.c", 16, ((idx) >= (10u) ? 0 : array[idx]));
537    }
538
539    return 0 ;
540  }
```

Content of `macro.i`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Preprocessor: #if and #endif

Conditional text inclusion:

```
'#if' <Expression>                    '#endif'
```

- `Expression`

  Consisting of constants and macro identifiers.

  - May contain usual C operators.
  - Undefined identifiers evaluate to `0`, except `true`.
  - Expressions may contain `'defined' <Identifier>` or `defined '(' <Identifier> ')'`:
    - Check whether a macro with the given identifier was defined.

- `#endif`

  Required to closes an `#if` (on a separate line).

- The text between the `#if` and `#endif` is only included when the expression evaluates to non-zero.

https://en.cppreference.com/w/c/preprocessor/conditional

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Preprocessor: `#ifdef` and `#ifndef`

Conditional text inclusion:

```
'#ifdef' <Identifier>
```

```
'#ifndef' <Identifier>
```

- `#ifdef`:
  Is the same as `'#if' 'defined' '(' <Identifier> ')'`.
- `#ifndef`:
  Is the same as `'#if' '!' 'defined' '(' <Identifier> ')'`.
- `#endif`
  Also required to close `#ifdef` and `#ifndef`.

https://en.cppreference.com/w/c/preprocessor/conditional

Preprocessor: `#error`

```
'#error' <Message>
```

- If the preprocessor encounters an `#error` directive.
  - It signals an error and prints the indicated message.
  - The compilation fails.
- This is useful to check for library features.
- The error directive is often guarded by an `#if`.

https://en.cppreference.com/w/c/preprocessor/error

TELECOM Paris

Example: #if and #error (1)

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

#if SIZE < 20
#error "The size is not sufficiently large."
#endif

#if __STDC_VERSION__ < 201710L
#error "Not the right version of C."
#endif

int main(int argc, char *argv[]) {
  printf("%d\n", SIZE);
  return EXIT_SUCCES;
}
```

Content of conditional.c.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: #if and #error (2)

```
tp-5b07-26:~/tmp> ls
conditional.c

tp-5b07-26:~/tmp> gcc -std=c11 -Wall -E -o conditional conditional.c
conditional.c:7:2: error: #error "The size is not sufficiently large."
 #error "The size is not sufficiently large."
  ^~~~~
conditional.c:11:2: error: #error "Not the right version of C."
 #error "Not the right version of C."
  ^~~~~
```


TELECOM Paris

Finding and Avoiding Bugs

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Assertions (1)

Any program is written with some **assumptions** in mind:

- How can you document these assumptions?
- How can you make sure that these assumptions are really satisfied?
  - **Runtime Assertions:**
    Check assumptions while the program executes.
    - Print an error message and maybe abort the program.
    - Always works.
    - May detect unforeseen glitches (e.g., bit flips).
    - Detects errors/violations when it is too late.
    - May slow down execution.
  - **Static Assertions:**
    Compiler checks assumptions and reports a compiler error.
    - Prevents errors before they happen.
    - Limited to assumptions that can be checked at compile time.
    - No runtime overhead.

Runtime Assertions

`'assert' '(' <Condition> ')'`

- Defined in `assert.h`.
- If the condition evaluates to zero:
  - `assert` prints an error message and aborts the program.
  - Otherwise the program continues normally.
- Implementation:
  - `assert` actually is a preprocessor macro.
  - It can be *deactivated* by defining `NDEBUG`.

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /*implementation defined*/
#endif
```

https://en.cppreference.com/w/c/error/assert

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Runtime Assertions (1)

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int safe_division(int dividend, int divisor) {
  assert(divisor != 0);
  return dividend/divisor;
}

int main(int argc, char *argv[]) {
  printf("%d\n", safe_division(25, 7));
  printf("%d\n", safe_division(25, 0));
  return EXIT_SUCCESS;
}
```

Content of `assertion.c`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Runtime Assertions (2)

```
tp-5b07-26:~/tmp> ls
assertion.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g assertion.c -o assertion

tp-5b07-26:~/tmp> ls
assertion  assertion.c

tp-5b07-26:~/tmp> ./assertion
3
assertion: assertion.c:7: safe_division: Assertion `divisor != 0' failed.
Aborted (core dumped)
```

NDEBUG is not defined by default – assertions are checked.

Example: Runtime Assertions (3)

The `gcc` compiler allows to define preprocessor symbols using the option `-D`:

```
tp-5b07-26:~/tmp> ls
assertion.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g -DNDEBUG assertion.c -o assertion

tp-5b07-26:~/tmp> ls
assertion  assertion.c

tp-5b07-26:~/tmp> ./assertion
3
Floating point exception (core dumped)
```

`NDEBUG` is explicitly defined using `-D` – assertions are not checked.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Static Assertions

```
'_Static_assert' '(' <Constant-Expression> (',' <Message>)? ')'
```

- Is a C language keyword.
- If the expression evaluates to zero (at compile time):
  - Triggers a compiler error.
  - Displays the optional error message.
  - Otherwise the compiler takes no action.
    (does not produce code for the static assertion)
- An alias `static_assert` is defined in `assert.h`.

https://en.cppreference.com/w/c/language/_Static_assert

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Example: Static Assertions (1)

```c
#include <assert.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// Checked by C compiler
static_assert(sizeof(int) == 8, "Expecting integer to have 8 bytes.");

// Checked by Preprocessor
// INT_MAX is defined by limits.h
#if INT_MAX != 0x7FFFFFFFFFFFFFFF
#error "Expecting integer to have 8 bytes."
#endif

int main(int argc, char *argv[]) {
  return EXIT_FAILURE;
}
```

Content of `static-assertion.c`.

TELECOM Paris

Example: Static Assertions (2)

```
tp-5b07-26:~/tmp> ls
static-assertion.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g static-assertion.c -o static-assertion
In file included from static_assertion.c:1:0:
static_assertion.c:7:1: error: static assertion failed: "Expecting integer to have 8 bytes."
 static_assert(sizeof(int) == 8, "Expecting integer to have 8 bytes.");
 ^
static_assertion.c:11:2: error: #error "Expecting integer to have 8 bytes."
 #error "Expecting integer to have 8 bytes."
  ^~~~~
```


TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

The Memcheck Tool

Memcheck is part of the Valgrind tool suite:

- It tracks all memory accesses, allocations, frees, …while executing a program.
- Based on these traces Memcheck reports:
  - Memory leaks (i.e., memory not freed).
  - Double freeing (memory freed twice).
  - Accesses to freed memory locations.
  - Accesses to uninitialized memory locations.
- It is good practice to check your code with Memcheck (or similar tools).
- **Usage on the command line:**

```
'valgrind' '--leak-check=full' <Program> <Command-line Arguments>?
```

https://valgrind.org/

TELECOM Paris

Example: Memory Errors (1)

```c
#include <stdio.h>
#include <stdlib.h>

void polluteStack(const char data) {
  char local[sizeof(int) + sizeof(void*)];
  for(int i = 0; i < sizeof(int) + sizeof(void*); i++)
    local[i] = data;
}
void *myAlloc(size_t size) {  // Produce 4 different memory errors.
  int b;                                      // Uninitialized local variable.
  void *result = b & 2 ? &b : malloc(size);   // Valid/Invalid pointer.
  if (b & 1)
    free(result);                             // Free valid/invalid pointer.
  return result;                              // Return a valid/invalid pointer.
}
int main(int argc, char *argv[]) {
  polluteStack(argc - 1);                     // Control content of variable b.
  int *array = myAlloc(3*sizeof(int));        // Get valid/invalid pointer.
  for(int i = 0; i <= 3; i++)
    array[i] = i;                             // Always produce an error.
}
```

Content of `memcheck.c`.

Example: Memory Errors (2)

```
tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g memcheck.c -o memcheck
tp-5b07-26:~/tmp> valgrind --leak-check=full ./memcheck
<snip>
==6631== Conditional jump or move depends on uninitialised value(s)
==6631==    at 0x4005A9: myAlloc (memcheck.c:12)
==6631==    by 0x4005E5: main (memcheck.c:18)
==6631==
==6631== Invalid write of size 4
==6631==    at 0x40060A: main (memcheck.c:20)
==6631==  Address 0x523504c is 0 bytes after a block of size 12 alloc`d
==6631==    at 0x4C332EF: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==6631==    by 0x40059C: myAlloc (memcheck.c:11)
==6631==    by 0x4005E5: main (memcheck.c:18)
==6631==
==6631== HEAP SUMMARY:
==6631==     in use at exit: 12 bytes in 1 blocks
==6631==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==6631==
==6631== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6631==    at 0x4C332EF: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==6631==    by 0x40059C: myAlloc (memcheck.c:11)
==6631==    by 0x4005E5: main (memcheck.c:18)
<snip>
```

Example: Memory Errors (3)

```
tp-5b07-26:~/tmp> valgrind --leak-check=full ./memcheck 1
<snip>
==6853== Conditional jump or move depends on uninitialised value(s)
==6853==    at 0x4005A9: myAlloc (memcheck.c:12)
==6853==    by 0x4005E5: main (memcheck.c:18)
==6853==
==6853== Invalid write of size 4
==6853==    at 0x40060A: main (memcheck.c:20)
==6853==  Address 0x5235040 is 0 bytes inside a block of size 12 free`d
==6853==    at 0x4C3451B: free (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==6853==    by 0x4005B6: myAlloc (memcheck.c:13)
==6853==    by 0x4005E5: main (memcheck.c:18)
==6853==  Block was alloc`d at
==6853==    at 0x4C332EF: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==6853==    by 0x40059C: myAlloc (memcheck.c:11)
==6853==    by 0x4005E5: main (memcheck.c:18)
==6853==
==6853==
==6853== HEAP SUMMARY:
==6853==     in use at exit: 0 bytes in 0 blocks
==6853==   total heap usage: 1 allocs, 1 frees, 12 bytes allocated
<snip>
```

Other Useful Tools

- **Debuggers:**
  - Allow to stop execution, single-step, inspect state …
  - Are unfortunately often cumbersome to use.
    - `gdb` is the golden standard on Linux.
    - You have seen `gdbgui` already.
    - Many IDE's come with graphical debuggers.
- **Profiler:**
  - Collect statistics on program execution.
  - Allow to spot performance issues.
  - Even harder to use than debuggers:
    - `valgrind`, `gprof`, `DTrace`, Linux `perf`, …
- **Test Coverage:**
  - Tools to do program testing.
  - Coverage indicates how *well* the code is tested.
    (e.g., statement coverage: every statement was tested at least once)
    - `gcov` for gcc, `SanitizerCoverage` for clang

Check Yourself! (1)

1. Suppose you write a piece of code that relies on the alignment of the following structure:

```
struct alignedStructure
{
  unsigned Address;
  char Payload[16];
};
```

   How can you express in your code that the alignment of this structure should be at least the size of the `int` type?

2. Is the requirement of the code from the previous question satisfied in all standard-compliant implementation? Why/Why not?

Check Yourself! (2)

3. What happens when you call `free` with a pointer that was not allocated on the heap (e.g., `NULL` or a pointer to a local variable on the stack)?
4. What happens when you deference a pointer to the heap that was freed?
5. How can you protect your code from such an error, or at least detect such a situation?

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Answers (1)

1. The size of the type `int` can be determined using the expression `sizeof(int)`, while the alignment of the structure can be determined using `_Alignof(alignedStructure)`. These are clearly expressions of the C language and have to be handled by the C compiler, thus you may express this assumption as a static assertion:

```
_Static_assert(_Alignof(alignedStructure) >= sizeof(int));
```

2. The alignment of the structure is compatible with the alignment of its members. So the alignment of the structure has to be at least the alignment of the `unsigned` member `Address`. In addition, the size and alignment of any signed integer type and its unsigned variant are identical. However, there is no link between alignment and size. Consequently, it is not guaranteed that this assumption is satisfied in all standard-compliant compilers.

Answers (2)

3. The result is undefined. On the lab machines the `free` function will in most cases detect the error, print an error message, and abort the program.
4. The result is undefined. On the lab machines in most cases nothing *suspicious* will happen immediately (there is probably no error message and the program continues *quietly*), after all the accessed address was valid just before the call to `free`. However, the address to which the pointer points might be reused by a subsequent call to `malloc`. Manipulating the memory through the initial pointer may then cause *unforeseeable* side-effects, e.g., the values of some object on the heap changes unexpectedly.
5. There is no good way to protect your C code against such errors – except switching to a safe language such as Rust or Java. Tools such as `valgrind` are able to detect such errors at runtime. Consequently, rigorous testing is in many cases the best way to find such errors.

Structuring a Code Base

Means to Structure Code

The C language provides little means to structure code:

- Regular C source files:
  - Just a sequence of declarations/definitions.
    (global variables, functions, derived types)
  - Variables/functions can be declared *private* using `static`.
- Header files:
  - Just a sequence of declarations.
    (derived types, variables/function with `extern`)
  - Everything in a header file is visible to other translation units.
- That's it.

Structuring a Code Base

- Use multiple C and header files.
- Group related algorithms/functions in a C source file (`.c`):
  - Function definitions.
  - Global variable declarations.
- Provide a matching header file (`.h`):
  - Types for data structures.
  - Declarations of variables/functions.
  - Preprocessor definitions and directives.
- Use a common prefix for global function, types, and variable names (e.g., `mylib_*`)
  - The C namespace is unique and shared by all translation units!
- Use sub-directories.
  - Regroup related header and source files.
  - Use sub-sub-directories.

Writing and Using Header Files

A header file defines an **interface** to an implementation:

- The interface allows to accesses functions/data structures.
- The interface is potentially (re)used several times.
  - Headers may include other headers.
  - C source files may include header files.

$\Longrightarrow$ **A C source file may include directly/indirectly the same header file multiple times.**

Example: Redundant Header Inclusion (1)

```c
typedef struct node node_t;
struct node {
  void *data;
  node_t *next;
};
```

Content of `linked-list3.h`.

```c
#include <stdio.h>
#include <stdlib.h>

#include "linked-list3.h"  // Including the same header ...
#include "linked-list3.h"  // twice, causes compiler errors.

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t node2 = {&data[2], NULL};
  node_t node1 = {(void*)&data[1], &node2};
  node_t node0 = {&data[0], &node1};

  for(node_t *tmp = &node0; tmp; tmp = tmp->next) {
    int *ptr = (int*)tmp->data;
    printf("%d\n", *ptr);
  }
  return EXIT_SUCCESS;
}
```

Content of `linked-list3.c`.

TELECOM Paris

Example: Redundant Header Inclusion (2)

```
tp-5b07-26:~/tmp> ls
linked-list3.c  linked-list3.h

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g linked-list3.c -o linked-list3
In file included from linked-list3.c:5:0:
linked-list3.h:2:8: error: redefinition of ‘struct node’
 struct node {
        ^~~~
In file included from linked-list3.c:4:0:
linked-list3.h:1:16: note: originally defined here
 typedef struct node node_t;
                ^~~~
```

Including the header twice leads to duplicate declarations and compiler errors.

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Protecting Headers

To avoid this problem:

- Add the following pattern at the beginning of the header file:

```
'#ifndef' <Header-identifier>
'#define' <Header-identifier>
```

- Close the `#ifndef` at the end of the header:

```
'#endif' // <Header-identifier>
```

- **Reasoning:**
  - Chose a unique header identifier.
    (usually by "mangling" the file name into a valid identifier, e.g., `stdio.h` → `_STDIO_H`)
  - On first inclusion:
    - The unique identifier was never defined.
    - The `#ifndef` will include the header's code …
    - … notably, the `#define` of the unique identifier.
  - In case of a redundant inclusion:
    - The unique identifier was defined by the first inclusion.
    - The `#ifndef` of subsequent inclusions will skip the header's code.

Example: Safe Redundant Header Inclusion (1)

```c
#ifndef LINKED_LIST4_H
#define LINKED_LIST4_H

typedef struct node node_t;
struct node {
  void *data;
  node_t *next;
};

#endif // LINKED_LIST4_H
```

Content of `linked-list4.h`.

```c
#include <stdio.h>
#include <stdlib.h>

#include "linked-list4.h"  // Including the same header ...
#include "linked-list4.h"  // twice, is no longer an issue.

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t node2 = {&data[2], NULL};
  node_t node1 = {(void*)&data[1], &node2};
  node_t node0 = {&data[0], &node1};

  for(node_t *tmp = &node0; tmp; tmp = tmp->next) {
    int *ptr = (int*)tmp->data;
    printf("%d\n", *ptr);
  }
  return EXIT_SUCCESS;
}
```

Content of `linked-list4.c`.

Example: Safe Redundant Header Inclusion (2)

```
tp-5b07-26:~/tmp> ls
linked-list4.c  linked-list4.h

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g linked-list4.c -o linked-list4

tp-5b07-26:~/tmp> ls
linked-list4  linked-list4.c  linked-list4.h

tp-5b07-26:~/tmp> ./linked-list4
4
55
-7
```

Compiling Multiple C Source Files

The `gcc` compiler accepts multiple C source files:

- Only a single `main` function is allowed.
- First each C source file is compiled separately:
  - Pretty much as if compiled manually.
  - For each C file an **object file** is produced.
    - Similar to an executable file.
    - Only contains code/data of current translation unit.
- Then all object files are **linked** together:
  - Take code/data of each object file.
  - Chain code/data together.
  - Result: the final **executable file**.

Example: Multiple Source Files (1)

```c
#ifndef LINKED_LIST5_H
#define LINKED_LIST5_H

typedef struct node node_t;
struct node {
  void *data;
  node_t *next;
};

#define emptyList NULL

extern node_t *pushFront(node_t *list, void *data);
extern void freeList(node_t *list);

#endif // LINKED_LIST5_H
```

Content of `linked-list5.h`.

Example: Multiple Source Files (2)

```c
#include "linked-list5.h" // Should always go first.
#include <stdlib.h>

node_t *pushFront(node_t *list, void *data) {
  node_t *new = malloc(sizeof(node_t));

  if (new) {
    new->data = data;
    new->next = list;
    return new;
  }
  else return NULL;
}
void freeList(node_t *list) {
  while(list) {
    node_t *tmp = list;
    list = list->next;
    free(tmp);
  }
}
```

Content of `liblinked-list5.c`.

```c
#include <stdio.h>
#include <stdlib.h>

#include "linked-list5.h"

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t *node2 = pushFront(emptyList, &data[2]);
  node_t *node1 = pushFront(node2, &data[1]);
  node_t *node0 = pushFront(node1, &data[0]);

  for(node_t *tmp = node0; tmp; tmp = tmp->next) {
    int *ptr = (int*)tmp->data;
    printf("%d\n", *ptr);
  }

  freeList(node0);

  return EXIT_SUCCESS;
}
```

Content of `linked-list5.c`.

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Example: Multiple Source Files (3)

Using the `-save-temps` option the `gcc` compiler keeps all intermediate files:[7]

```
tp-5b07-26:~/tmp> ls
liblinked-list5.c  linked-list5.c  linked-list5.h

tp-5b07-26:~/tmp> gcc -save-temps -Wall -pedantic -std=c11 -O0 -g \
                      liblinked-list5.c linked-list5.c -o linked-list5

tp-5b07-26:~/tmp> ls
liblinked-list5.c  liblinked-list5.o  linked-list5    linked-list5.h  linked-list5.o
liblinked-list5.i  liblinked-list5.s  linked-list5.c  linked-list5.i  linked-list5.s

tp-5b07-26:~/tmp> ./linked-list5
4
55
-7
```

[7] Usually not needed in real toolchains. Used here for demonstration purposes only.

TELECOM Paris
Institut Polytechnique de Paris

Compilation Steps of `gcc`

The `gcc` compiler produces several intermediate files:

- Per C source file:
  - **Preprocessor:** (`.c` files ⇒ `.i` files)
    C source file after preprocessing.
  - **C Compiler:** (`.i` ⇒ `.s`)
    Assembly code after compilation.
  - **Assembler:** (`.s` ⇒ `.o`)
    Object file containing machine code.

- **Linker:**
  Takes all `.o` files and *links* them together — producing the **executable file**.

Stopping after Compilation Steps

One can stop `gcc` after each compilation step using command-line options:

- Per C source file:
  - **Preprocessor:** `-E`
    Stop `gcc` after the preprocessor (`.c` ⇒ `.i`).
  - **C Compiler:** `-S`
    Stop after the C compiler (`.c` ⇒ `.s`).
  - **Assembler:** `-c`
    Stop after the assembler (`.c` ⇒ `.o`).
- Any of the above options disables the linker.
- No need to specify the name of the output (`-o` option not needed).

Example: Manually Compiling and Linking

Object files can be produced explicitly by gcc using the -c option:

```
tp-5b07-26:~/tmp> ls
liblinked-list5.c  linked-list5.c  linked-list5.h

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g liblinked-list5.c -c
tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g linked-list5.c -c

tp-5b07-26:~/tmp> ls
liblinked-list5.c  liblinked-list5.o  linked-list5.c  linked-list5.h  linked-list5.o

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g liblinked-list5.o linked-list5.o -o linked-list5

tp-5b07-26:~/tmp> ls
liblinked-list5.c  liblinked-list5.o  linked-list5  linked-list5.c  linked-list5.h  linked-list5.o

tp-5b07-26:~/tmp> ./linked-list5
4
55
-7
```

Compilation in Big Projects

Nobody compiles big projects manually using `gcc` directly:

- Instead **build tool** are used:
  - Allow to specify which output files to produce.
  - Allow to specify on which input files each output file depends.
  - Computes automatically which files to compile and how.
    - Compile only code that has changed.
    - Compile code in **parallel**.
  - **Examples:**
    - `make` (old) or `cmake` (modern).
    - Build systems of IDEs (Visual Studio/Code, …).
- Often use **libraries**:
  - Allow to structure code.
  - Allow to reuse common code.
    (e.g., graphical interfaces, data structures, …)

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Libraries: Revisiting the Standard C Library

- How does the compiler find header files?
  - Usually stored in an **implementation-defined** directory.
  - Lab machines: `/usr/include/`.
- Where is the machine code (e.g., of `printf`)?
  - Stored in **implementation-specific** library files.
  - Lab machines offer two options:
    - Static linking: `/lib/x86_64-linux-gnu/libc.a`
    - Dynamic linking: `/lib/x86_64-linux-gnu/libc.so.6`
- `gcc` searches these files automatically.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Library Files

- Static libraries are called **archives**:
  - Archive files have the suffix `.a`.
  - A collection of **object files** with a **symbol index**:
    - All variables/functions appear in the index.
- Dynamic libraries:
  - More like an executable file (without `main` though).
  - Contain all data/code of the library.
  - Loaded at the same time as the executable file.
  - Revisited in more detail in Part 3.
- **Library file names:**
  - Library file names **always** start with the prefix `'lib'`.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Symbols of `libc.a`

```
tp-5b07-26:~/tmp> nm /lib/x86_64-linux-gnu/libc.a
<snip>
fprintf.o:
<snip>
                 U __vfprintf_internal
0000000000000000 T fprintf

printf.o:
<snip>
                 U __vfprintf_internal
0000000000000000 T printf
                 U stdout
<snip>
stdio.o:
0000000000000000 D stderr
0000000000000010 D stdin
0000000000000008 D stdout
<snip>
```

Creating a Static Library Archive

- Create a (or several) `.h` header file(s):
  - Define the interface to the library.
  - Include types, variable and function declarations (`extern`).
- Create multiple C source files:
  - Regroup function definitions and variable declarations (without `extern`).
  - Use `static` keyword to hide variables/functions when needed.
- Compile C source files:
  - Producing **object files** (`-c` option).
  - Build the archive using the following command pattern:

```
'ar' 'r' <Library-name> <Object-files>
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Example: Static Library Archive (1)

```c
#ifndef LINKED_LIST6_H
#define LINKED_LIST6_H

typedef struct node node_t;
struct node {
  void *data;
  node_t *next;
};

#define emptyList NULL

extern node_t *pushFront(node_t *list, void *data);
extern void freeList(node_t *list);

#endif // LINKED_LIST6_H
```

Content of `linked-list6.h`.

TELECOM Paris INSTITUT POLYTECHNIQUE DE PARIS

Example: Static Library Archive (2)

```c
#include "linked-list6.h" // Should always go first.
#include <stdlib.h>

node_t *pushFront(node_t *list, void *data) {
  node_t *new = malloc(sizeof(node_t));

  if (new) {
    new->data = data;
    new->next = list;
    return new;
  }
  else return NULL;
}
```

Content of pushfront.c.

```c
#include "linked-list6.h"  // Should always go first.
#include <stdlib.h>

void freeList(node_t *list) {
  while(list) {
    node_t *tmp = list;
    list = list->next;
    free(tmp);
  }
}
```

Content of freelist.c.

Example: Static Library Archive (3)

```
tp-5b07-26:~/tmp> ls
freelist.c  linked-list6.h  pushfront.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g -c pushfront.c freelist.c

tp-5b07-26:~/tmp> ls
freelist.c  freelist.o  linked-list6.h  pushfront.c  pushfront.o

tp-5b07-26:~/tmp> ar r liblinked-list6.a pushfront.o freelist.o
ar: creating liblinked-list6.a

tp-5b07-26:~/tmp> nm liblinked-list6.a

pushfront.o:
                 U malloc
0000000000000000 T pushFront

freelist.o:
                 U free
0000000000000000 T freeList
```


TELECOM Paris

Using (external) Libraries

The `gcc` compiler needs to find the library/header files:

- **Include path:**
  Directories where the compiler searches for header files.
  - `#include <>`:
    Searches only in *standard include directories*.
  - `#include ""`:
    Searches first in the directory of the currently compiled C source file, then in standard directories.
  - You can specify additional directories using the option `-I`:

```
'gcc' <options> ('-I' <Include-directory>)* <Source-files>
```

- **Library path:**
  Directories where the compiler searches for library files.
  - `gcc` by default searches *standard library directories*.
  - You can specify additional directories using the option `-L`:

```
'gcc' <options> ('-L' <Library-directory>)* <Source-files>
```


TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Using (external) Libraries (2)

Telling `gcc` to link against a library:

- **Library file names:**
  - Recall: library file names always start with `'lib'`.
- To tell `gcc` to search a library use the `-l` option:
  - *Without* the prefix `'lib'`, `gcc` adds it automatically.

```
'gcc' <options> <Source-files> ('-l' <Library-name>)*
```

Example: Using a Static Library Archive (1)

```c
#include <stdio.h>
#include <stdlib.h>

#include "linked-list6.h"

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t *node2 = pushFront(emptyList, &data[2]);
  node_t *node1 = pushFront(node2, &data[1]);
  node_t *node0 = pushFront(node1, &data[0]);

  for(node_t *tmp = node0; tmp; tmp = tmp->next) {
    int *ptr = (int*)tmp->data;
    printf("%d\n", *ptr);
  }

  freeList(node0);

  return EXIT_SUCCESS;
}
```

Content of `linked-list6.c`.

Example: Using a Static Library Archive (2)

```
tp-5b07-26:~/tmp> ls ./liblinked-list/
liblinked-list6.a  linked-list6.h

tp-5b07-26:~/tmp> ls
linked-list6.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g -I./liblinked-list/ -L./liblinked-list/ \
                  -o linked-list6 linked-list6.c  -llinked-list6

tp-5b07-26:~/tmp> ls
linked-list6  linked-list6.c

tp-5b07-26:~/tmp> ./linked-list6
4
55
-7
```

Check Yourself!

1. List the kind of files that may contain machine code/binary data.
2. Which option is needed to ask `gcc` to produce an object file?
3. Which compilation phase actually produces the object file?
4. Which option is needed to indicate to `gcc` that your program requires a library?

TELECOM Paris

INSTITUT POLYTECHNIQUE DE PARIS

Answers (1)

1. In the lecture you have seen several kinds of files that may hold machine code:
   - Executable files
   - Object files
   - Library archives
   - Dynamic libraries
2. The `gcc` compiler produces object files when the `-c` option is provided. In this case the file name of the produced object file is automatically derived from the name of the C source file.

TELECOM Paris
INSTITUT POLYTECHNIQUE DE PARIS

Answers (2)

3. The compilation proceeds in multiple phases using the following tools:

   a. The preprocessor.
   b. The C compiler.
   c. The assembler.
   d. The linker.

   The preprocessor applied text substitutions, and thus produces *just* C source code. The C compiler translates the C source code to assembly language. The assembler translates the assembly language to machine code – **producing an object file**. The linker combines all of the program's object files, along with additional library files, to produce the final executable file.

   The answer is thus: the assembler.

4. You can ask `gcc` to search for a library and link it to your program using the `-l` option. Recall, the filename of the library always starts with the prefix `'lib'`, which is dropped for the `-l` option.

   You may need in addition supply the `-I` and `-L` options, in order to make sure that `gcc` finds the header files of the library as well as the library files (an archive or dynamic library).

Lab Exercises

Polish code from previous lab session:

- Check memory operations with `valgrind`
- Restructure code:
  - Write header files.
  - Split code into multiple C source files.
  - Use sub-directories.
- Regroup reusable code into a static library.
  - Create a static library.
  - Link against that library.

Credits

- Warning icon created by Vectors Market - Flaticon

TELECOM Paris