Part 2.1 - The C Language

ECE_3TC31_TP/INF107

Florian Brandner

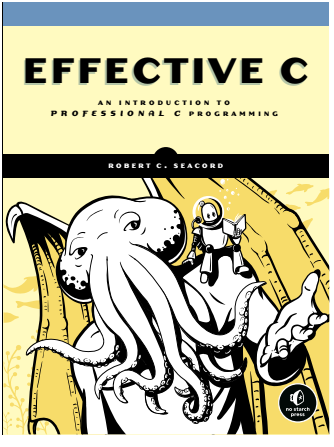
2024

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
- **CPP Reference:**
  https://en.cppreference.com/w/c



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

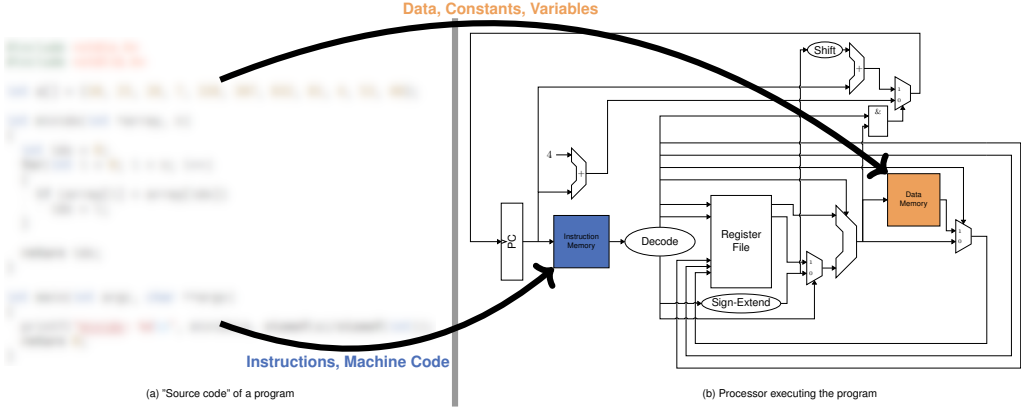
202x: Upcoming ISO standard (C23) currently under development

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.

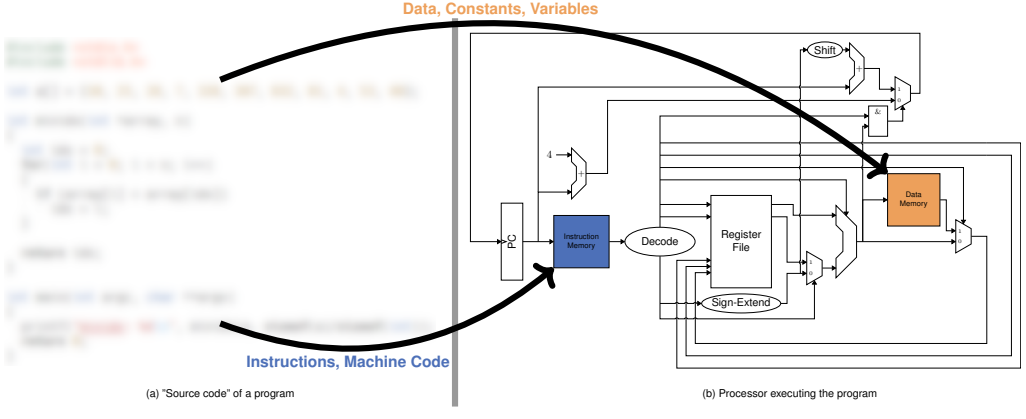
Compilers - From source code to machine code

From source code to machine code (1)



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

From source code to machine code (1)

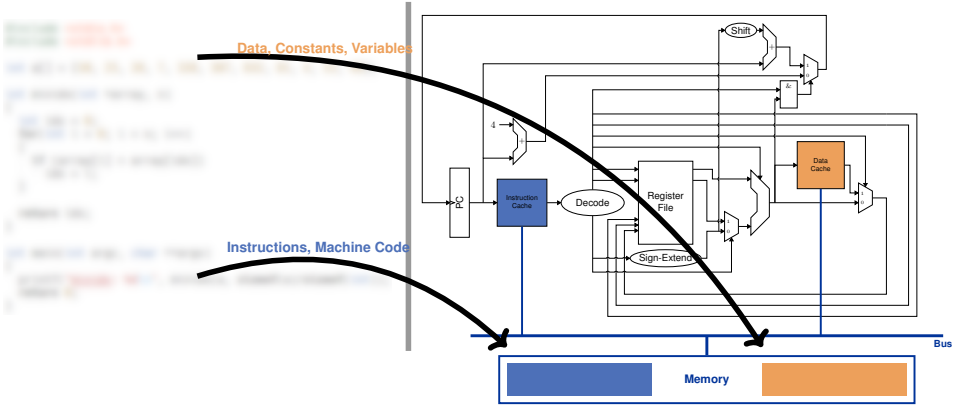


(a) "Source code" of a program

(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

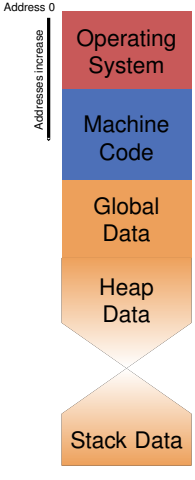
(b) Processor executing the program

… 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

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)

```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`.

- 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).

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
```

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

What is a Type?

Types are a common concept in programming languages:

- A type specifies the **set of values** 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.

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:

```
_Bool done = false;          Initializes the variable done to false.
_Bool isfalse = 0;           Initializes the variable isFalse to false.
_Bool isTrue = 5;            Initializes the variable isTrue to true.
_Bool isTrueToo = true;      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]

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

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.

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`

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`

Examples: Integer Types and Literals

Essential notations you have to know:

`unsigned char c = 225;`	Initializes the variable `c` to 225.
`int i = 512;`	Initializes `i` to 512.
`unsigned ui = 5u;`	Initializes `ui` to 5 (using unsigned literal suffix 'u').
`signed hex = 0x10;`	Initializes `hex` to 16 (using base 16).

Other notations you might see:

`short octal = 010;`	Initializes `octal` to 8 (using base 8).
`long int li = 0x20000010l;`	Initializes `li` to $536\,870\,928$ (`l` suffix and base 16).
`long long lli = 0x202000001011l;`	Initializes to $137\,975\,824\,400$ (`ll` suffix and base 16).

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

Floating-Point Types and Literals

`float`	Single-precision, usually 32 bit.
`double`	Double-precision, usually 64 bit.
`long double`	Extended-precision, usually 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 $2.0e308$.

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

Character Types and Symbols

`char`	Equivalent either to `signed char` or `unsigned char`, usually 8-bit ASCII value.
`char16_t`	A Unicode character (in the UTF-16 encoding).
`char32_t`	A Unicode character (UTF-32).

- Examples:

`char c = 'a';`	Initializes the variable `c` to the symbol `a` (97 decimal).
`char clf = '\n';`	Initializes `cf` to the line feed symbol (see next slide).
`char16_t c16 = u'β';`	Initializes `c16` to the symbol β (UTF-16 prefix, little beta).

https://en.cppreference.com/w/c/language/character_constant
https://en.wikipedia.org/wiki/ASCII

Character Escape Sequences

Escape Sequence	Description	ASCII Code	Escape Sequence	Description	ASCII Code
`\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 hi[] = "Hello World\n";
```
Initializes variable `hi` to the given string.

```
char beta1[] = u8"Greek beta: β";
```
Initializes `beta1`, using UTF-8 encoding.

```
char16_t beta2[] = u"Beta: \u0387";
```
Initializes `beta2`, using UTF-16 encoding.

```
char32_t german[] = U"German S: ß";
```
Initializes `german`, using UTF-32 encoding.

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

Global Declarations and Definitions

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 sequences 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 are optional and 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

`int counter = 0;`	Declare the variable `counter` (with initializer).
`const short constant = 27;`	Declare `constant` as `const`, i.e., its value is not supposed to change during execution.
`extern unsigned elsewhere;`	Declare `elsewhere` with storage class `extern`.
`static char private = 'p';`	Declare `private` with storage class `static`.
`short data[100];`	Declare `data` as an array of 100 `short` values, stored consecutively in memory.
`int init[3] = {0, 1, 2};`	Declare `init` as an array of 3 `int` values (initialized to 0, 1, and 2 respectively.)
`char msg[] = "Hello World";`	Declare `msg` as an array of characters (size derived).

Examples: Global Function Declarations

`void foo();`	Declare the function `foo`, which does not return anything and takes no argument.
`int bar(char a, short b);`	Declare `bar`, taking two arguments and returning an `int` value.
`extern char elsewhere(int, int b);`	Declare `elsewhere` with storage class `extern`, and two arguments (one without name).
`static void priv(int a, int b);`	Declare `priv` with storage class `static`, does not return anything, takes two arguments.

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 (≠ *declarations*) consists of four parts:

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

Resembles a function declaration:

- **Storage class**, **Qualifiers**, **Type**, **Declarator**:
  Same as for function declarations.
- **Body:** (← was missing in declarations)
  The code of the function enclosed in curly braces.

Example (our previous `main` function):

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

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

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!

```
1
2  const char message[] = "Hello World";
3
4  int main(int argc, char *argv[])
5  {
6    int data = 0;
7    {
8      static const char message[] = "Me First";
9      printf("%s\n", message);
10   }
11   printf("%s\n", message);
12   return EXIT_SUCCESS;
13 }
```

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

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
```

Expressions (Quick)

Expressions

Compute a single value from:

- Constants
  Same notations as seen before when we introduced types.
- Variable values
  Referred to 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

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

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)

Precedence	Operator	Description	Associativity
6	<	Less	Left
	<=	Less-equal	
	>	Greater	
	>=	Greate-equal	
7	== !=	Compare for equality Compare not equal	Left
8	&	Bitwise AND	Left
9	^	Bitwise XOR	Left
10	|	Bitwise OR	Left
11	&&	Logical AND (short-circuit)	Left
12	||	Logical OR (short-circuit)	Left
13	? :	Conditional Operator	Right
14	=	Assignment Operator	Right

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

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`

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:

```
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

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

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)

This `for` loop:

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

is hence analogous to the `while` loop:

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

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 (don't do that).

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

A First C Program: Division by Subtraction

```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`.

Let's Run our Division Program

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
```

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` on the command line results in
  `argc`: 3
  `argv[0]`: `"./division"`
  `argv[1]`: `"one"`
  `argv[2]`: `"2"`

The C Standard Library

The C Standard Library

The C standard library (AKA `libc`) 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

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.
- Example:
  `#include <stdio.h>` - Include declarations of file, input, and output library.

Example: Header File

Here is an excerpt from the `libc` 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>
```

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:

```
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 will not cover them in this course.[6]
  - `format` specifies how to display the other parameter values.
- **Example:** `printf("A number: %d\n", 5)`

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

[6] See https://en.cppreference.com/w/c/variadic to learn more about variadic functions.

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`.

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
```

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?

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.

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[7]
    (compute the primes up to 100, print integers on screen)
  - Insertion sort
    (sort floating-point numbers in an array, print floats on screen)

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

Lab Exercises (2)

How to read/use the slides:

- Use the slides as a **reference**:
  - Lookup how to declare variables or functions.
  - Lookup how to define functions.
  - Lookup how to compile programs.
  - Lookup further documentation using the embedded links.
  - In the lab:
    - Try things on your own.
    - Try to find answers yourself in the slides (see above).
    - Ask the teacher, if you cannot find the answer within a couple of minutes.
- Use the slides to prepare for the exam:
  - Go through the “Check Yourself” slides.
  - Focus on concepts (types, scopes, linkage, precedence, …).
  - Syntax is less important.