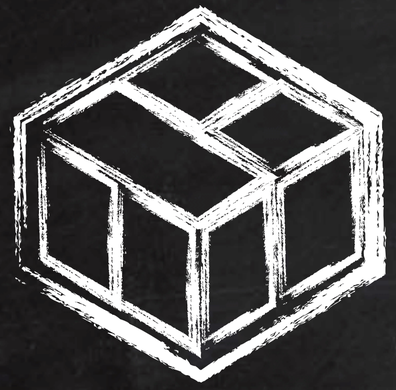


Architecting and Engineering Main Memory Database Systems in Modern C

Chapter 2
The C Language

Marcus Pinnecke, M.Sc.
Research associate / Working Group Database & Software Engineering
Institute of Technical and Business Information Systems (ITI)

OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG


Choosing a Programming Languages
THE RIGHT TOOL FOR THE TASK
VENDOR-LOCKIN VS OPEN
FAVORED SEMANTICS
THIRD-PARTY FUNCTIONALITY SUPPORT
PRODUCTIVE AND FAST DEVELOPMENT
PROJECT CONSTRAINTS
FAVORED SYNTAX
REALTIME REQUIREMENTS
APPLICATION VS SCRIPT VS SERICE VS SYSTEM
LEARNING CURVE
DOMAIN
EXPERIENCE & SKILLS
GOOD STANDARD LIBRARY
ECO-SYSTEM AND TOOL SUPPORT
LOW-LEVEL CONTROL
DESKTOP VS SERVER VS MOBILE
DESKTOP VS SERVER VS MOBILE
PROGRAMMING PARADIGM
LANGUAGE COMPLEXITY
LANGUAGE AGE
PORTABILITY (WEB, WIN, LINUX, CROSS PLATTFORM)
COMMUNITY SUPPORT
LANGUAGE FEATURES
Basic
Ruby
C
Dart
Python
Perl
Rust
Pascal
Kotlin
ActionScript
Scala
TypeScript
CUDA C
Machine Code
PL/SQL
C++
C#
bourne shell
R
QBasic
Java
Go
JavaScript
Windows PowerShell
Swift
Bytecode
Groovy
OpenCL
Assembler
Haskell
Swift
PHP
Object Pascal
Object Pascal
Lua
Lisp
PostScript
Objective-C
Smalltalk

Why did *you* choose to visit a course
having the words *engineering* and *C* in the title?

If we Choose C for DB Dev (Lecturers Personal Impression)

FAVORED SYNTAX

FAVORED SEMANTICS

THIRD-PARTY FUNCTIONALITY SUPPORT

VENDOR-LOCKIN VS OPEN

PROJECT CONSTRAINTS

DOMAIN

REALTIME REQUIREMENTS

PRODUCTIVE AND FAST DEVELOPMENT

EXPERIENCE & SKILLS

APPLICATION VS SCRIPT VS SERICE VS SYSTEM

LOW-LEVEL CONTROL

LEARNING CURVE

LANGUAGE COMPLEXITY

GOOD STANDARD LIBRARY

DESKTOP VS SERVER VS MOBILE

PROGRAMMING PARADIGM

ECO-SYSTEM AND TOOL SUPPORT

COMMUNITY SUPPORT

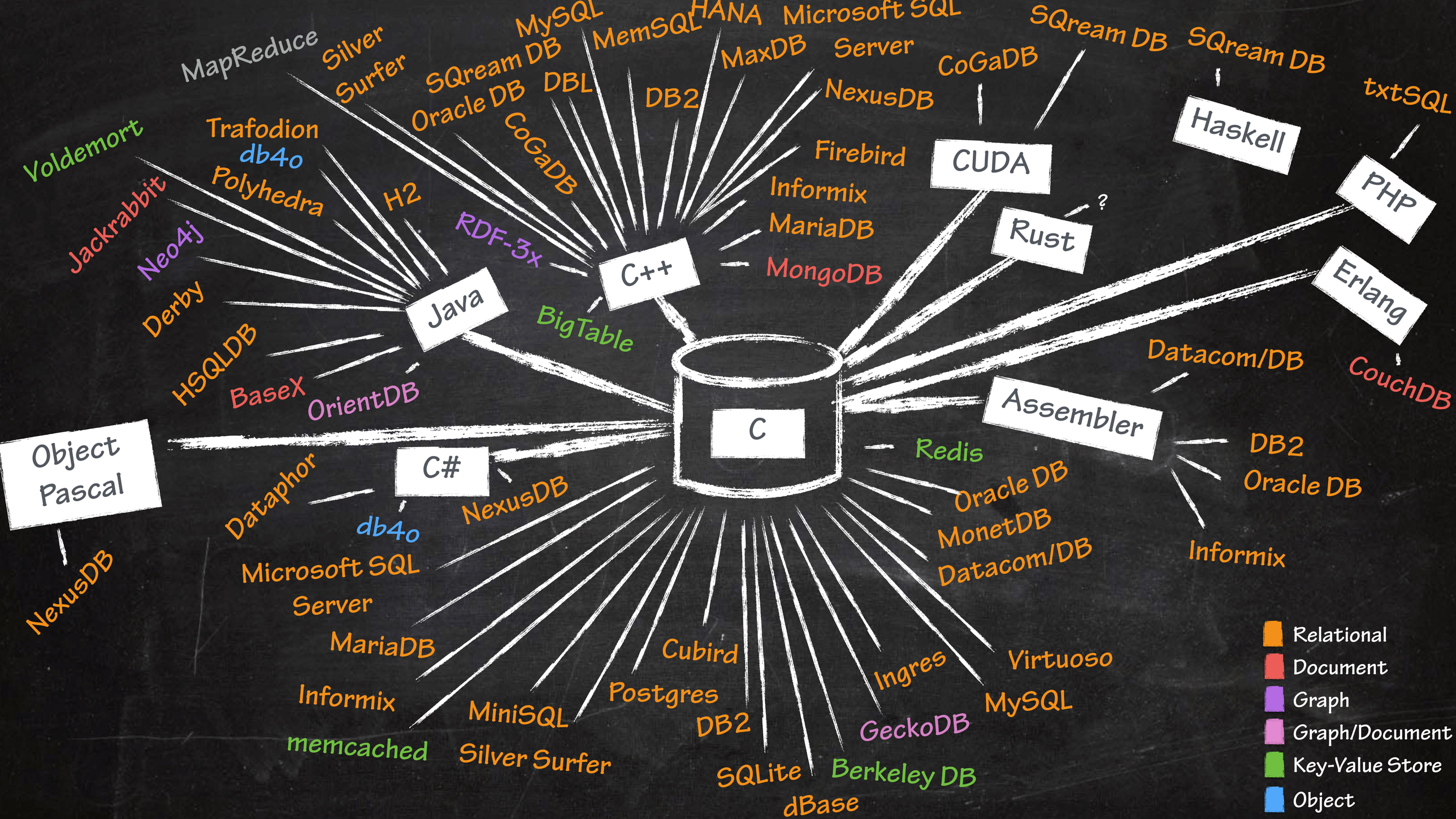
DESKTOP VS SERVER VS MOBILE

LANGUAGE AGE

LANGUAGE FEATURES

PORTABILITY (WEB, WIN, LINUX, CROSS PLATTFORM)

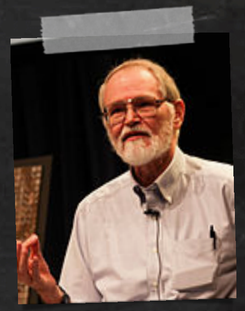
!


C
Java
C++
C#
Object Pascal
Assembler
Haskell
PHP
Erlang
CUDA
Rust
?
MapReduce
Silver Surfer
MySQL
SQream DB
Oracle DB
DBL
CoGaDB
MemSQL
HANA
DB2
MaxDB
Microsoft SQL Server
NexusDB
CoGaDB
SQream DB
SQream DB
txtSQL
Voldemort
Trafodion
db4o
Polyhedra
Jackrabbit
Neo4j
Derby
HSQLDB
BaseX
OrientDB
H2
RDF-3x
BigTable
Firebird
Informix
MariaDB
MongoDB
Datacom/DB
CouchDB
Redis
Oracle DB
MonetDB
Datacom/DB
DB2
Oracle DB
Informix
Dataphor
db4o
NexusDB
Microsoft SQL Server
MariaDB
Informix
memcached
MiniSQL
Silver Surfer
Cubird
Postgres
DB2
SQLite
dBase
Berkeley DB
Ingres
GeckoDB
Virtuoso
MySQL
NexusDB
Relational
Document
Graph
Graph/Document
Key-Value Store
Object

INTRODUCTION

Design Goals of C

Kenneth Thompson,
Creator of C



Brian W. Kernighan,
Creator of C

Dennis Ritchie,
Creator of C

- Small language complexity: limit to most important control structures and data types
- Efficient compiler: which takes little time for translation
- Minimal runtime support: compiler and platform-specific standard library & runtime
- Platform portability*: abstraction from hardware and operating system
- Highly efficient programs: compiling to programs with high execution performance and small memory footprint

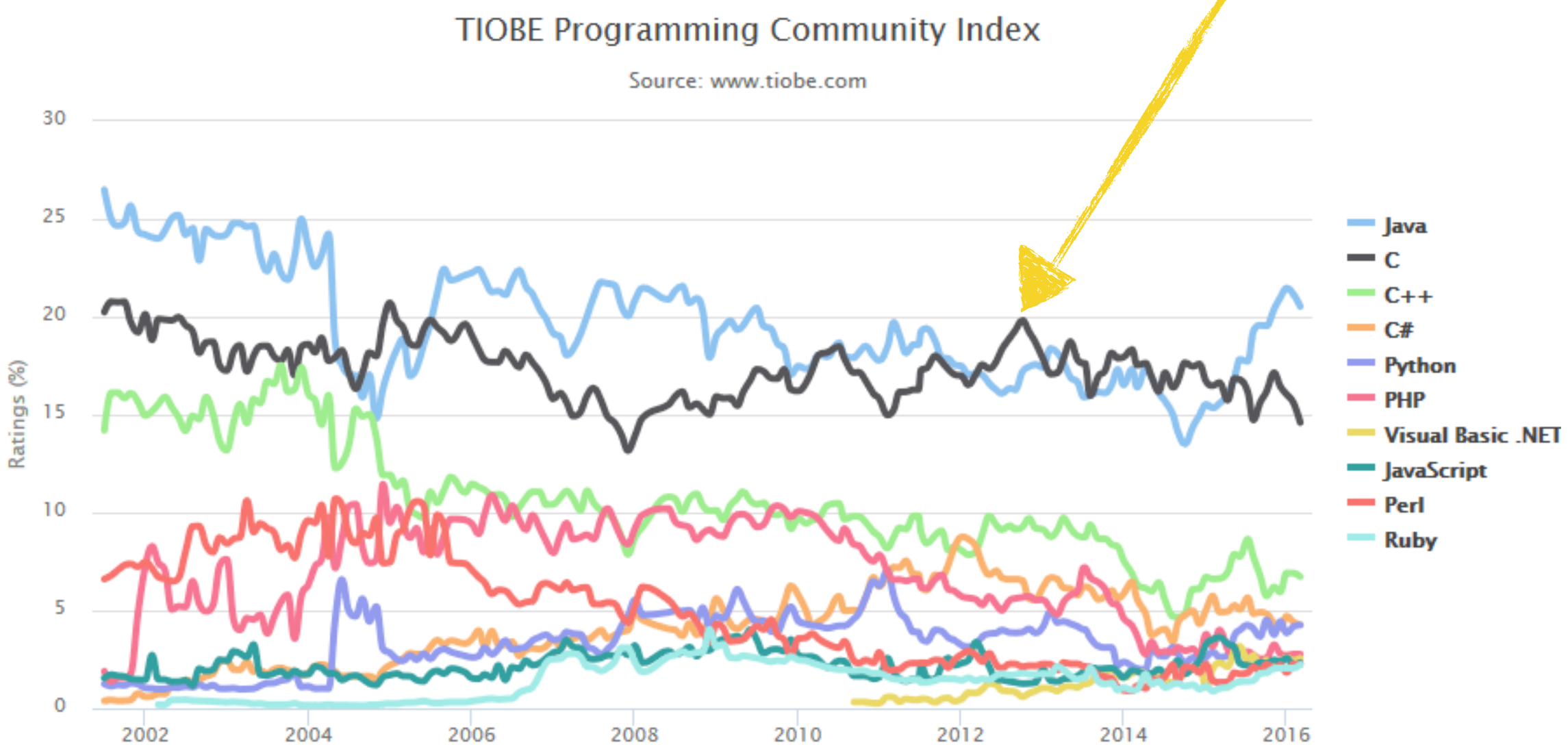
* was not the a primary design goal in advanced but turned out to be reasonable

C Programming Language

- General-purpose programming language
  - structured: built-in support of sub-routines („functions“), loops, and block structures
  - imperative: computation is expressed as statements changing program state
  - procedural: problem solving step by step by a particular order on procedure calls
  - static variable scope: names refers to its local lexical environment
  - supports recursion: problem solution depends on solutions to smaller instances of same problem
  - static type system: types of variables are known at compile time (weak enforced)
- High-level programming language which is often perceived as „low-level“
  - characters, numbers and addresses are first-class citizen
    - composite objects aren’t
  - sometimes referred as „portable assembler“
- often perceived as system programming language (database, operating systems)
- development driven by first UNIX operating system implementation
- It’s around for 40 years, and stable in its popularity

The Market of Programming Languages

according to the TIOBE index, C is both popular and stable in its popularity



Notable Properties

- C implementations must provide functions to enable „standard“ functions that are *not* built-into the C language itself (due to goal of small and portable language):
  - operation to composite objects (strings, lists, arrays)
  - storage definition beyond the built-in static one (e.g., heap or garbage collection)
  - input/output functionalities, file access methods
- Most other language compilers are required to provide call-into/out-call to C
- Deterministic destruction of objects
- Explicit dynamic memory allocation and deallocation
- C comes with a preprocessor that is not part of the C language itself
  - file inclusion
  - macro substitution
  - conditional compilation
- Since C11, multi-threaded control flow is part of the language per design

Programming in C

PROs

- C is small, and clean: limited number of keywords, and compact language concepts support understanding of third-party code.
- Modular code: separate compiling and linking of source files including basic forms for information hiding
- Slow evolution of an old language: new standards does not radically rewrite C and, hence, do not break existing code
- Time and memory efficient programs: programs in C are compiled to native code, fast, and have little memory overhead.
- C code is portable: as long as one considers the C language specification and puts some effort in cross-platform design, unmodified C code can be compiled to all major platforms*

CONs

- Strings are a missing concept: there is no explicitly concept of string. Instead, strings are implicitly available by the concepts of null terminated sequences of characters.
- Security flaws: although minimal in its design, the correct use of C is hard; origin of attack surfaces
- Missing composite object library: C is *not* shipped with most important data structures (lists, maps, …) on board. Sometimes „Reinventing the wheel“
- No built-in boolean values: „true“ and „false“ are defined constants but not a language concept. Sometimes confusing or problems during runtime.
- Potential memory leaks: Due to manual explicit management of dynamic memory, memory leaks can occur when incorrectly implemented

- Almost no limitation on what one can do: C has no built-in mechanism preventing you from doing things (cf. Java references vs. C pointers). But this power comes with great responsibility.

* The code is portable while the binary is not. However, compiler-specific differences in the implementation of the C specification, operating-system-specific interpretation of the POSIX standard, vendor-specific extension, platform-depending libraries etc. limit out-of-the-box portability of C programs.

History of C

1978 — K&R-C

First implementation by Brian W. Kernighan und Dennis Ritchie, and promoted in the famous book „The C Programming Language (1st Edition)“

1987 — X/Open-C

First attempt to standardize the language

1989 1990 — ANSI C aka C89 aka ISO C90

First standardized version of C (by ANSI, later by ISO).

New language features: function prototypes, possibility to declare constant values, more powerful preprocessor, `void` for empty function parameter list and as function return type for function who return nothing, new keywords `const`, `volatile` and `signed`, added support for characters with more than 8bit width

First specification of the C standard library.

1995 — ISO C95

Error corrections and some new features: alternate operator spelling, and standard macro `__STDC_VERSION__`

History of C

1999

ISO C99

New features: `inline` functions, variable declaration inside `for` statement, complex number via `_Complex` type, new type `long long` with 64 bit size at least, variable length arrays (VLA), boolean type `_Bool`, more math function in standard library, new keyword `restrict`, declaration anywhere inside blocks, restriction to use implicit function declaration and force function return type, variable length preprocessor macros, line comment `//` added,

2011

ISO C11

Native support of multi-threading (cf. standard library `<threads.h>`, `<stdatomic.h>`), control over data alignment & structure padding & packing, Unicode support (new types `char16_t`, `char32_t` and `<uchar.h>` in standard library), security improvements on the standard library, added exclusive read/write mode for file I/O, generic datatypes via `_Generic`

Quotes from Bjarne Stroustrup

Bjarne Stroustrup,
Creator of C++

C is flexible. It is possible to apply C to most every application area and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written

C is efficient. The semantics of C are „low level“; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or programmer to efficiently utilize hardware resources for C programs

Quotes from Bjarne Stroustrup

Bjarne Stroustrup,
Creator of C++

C is available: Given a computer, whether the tiniest micro or the largest super-computer, chances are that there is an acceptable quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. Libraries and support tools are also available, so that a programmer rarely needs to design a new system from scratch

C is portable: A C program is not automatically portable from one machine (and operating system) to another, nor is such a port necessarily easy to do. It is however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependences is typically technically and economically feasible

Tipp

Super useful resource

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

HANDS-ON C

Hello World in C

- Most simple beginners program outputting „Hello World“
- Historic one (K&R C dialect), around 1978

material/Chapter 02/00 Hello World/main_KandR_C.c

```
main() {
    printf("Hello, World!\n“);
    return 0;
}
```

- Standard today

material/Chapter 02/00 Hello World/main_C11.c

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0; // optional
}
```

- Compile source file and run the generated assembler out file

```
$ cc main_C11.c && ./a.out
Hello, World!
```

Inspect Hello World (1)

Program Entry for C Programs

```
return-type main( main parameter declaration )
{
    printf("Hello, World!\n");
    return return-value;
}
```

- Any executable written in C must contain the program entry point function `main`
  - normally function naming is up to programmer with exception of `main`
  - program execution starts at `main` function
  - `return-type` specifies the data type of the returned value (if any) when `return` statement is reached and program exits.
    - When left out, `int` type is implicitly be stated.
    - Specific for `main` : return type must be either
      - `void` if no value is returned, or
      - `int` a signed number indicating program state after termination

Inspect Hello World (2)

Program Entry for C Programs

```
return-type main( main parameter declaration )
{
    printf("Hello, World!\n");
    return return-value;
}
```

- `return return-value;` when control-flow reached this statement, a function returns the control-flow to the caller.
  - Specific for `main` : the program exists
  - `return` must not explicitly stated
    - but somehow required since C90 to avoid undefined behavior
  - In case of non-empty return type (i.e., `int`), the following value can be returned
    - zero (macro `EXIT_SUCCESS`): successful exit
    - non-zero (macro `EXIT_FAILURE`): abnormal program termination

Inspect Hello World (3)

Program Entry for C Programs

```
return-type main( main parameter declaration )
{
    printf("Hello, World!\n");
    return return-value;
}
```

- `main parameter declaration` arguments given to a function by the caller.
  - Specific for `main`: program arguments for the executable
  - Two parameter declaration for main:
    - `void` indicates that program arguments are ignored
    - `int argc, char **argv` (alternatively `int argc, char *argv[]`)
      - argument count `argc` is number of program arguments
        - greater or equal 1
      - argument vector `argv` is list of strings where *i*-th string is *i*-th argument
        - first string is always absolute path to executable

Example: material/Chapter 02/00 Hello World/main_with_args.c

Inspect Hello World (4)

Program Entry for C Programs

```
return-type main( main parameter declaration )
{
   printf("Hello, World!\n");
   return return-value;
}
```

- `printf("Hello, World!\n");` is a call to a built-in („standard library“) function
  - it prints „Hello, World!“ followed by a new line to the output stream (stdout)
  - (Standard) library functions can be used „out of the box“
  - function definition (i.e., implementation) is in the (standard) library itself
  - function declaration (announcing that „there exists a specific function `printf` having that particular argument list“) is done in `stdio.h` (header) file
  - functions are made available in the program via (preprocessor) file inclusion

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

Before you Write C code, consider what you not have to write since it's built-in.

The C Standard Library

- Set of well-specified built-in functionality
- C language implementation-independent
- First specified with ANSI C
- Contains most fundamental macros, types, and functions
- Particular functionality is added to a program by
  1. including particular C standard libraries header files (see later)
     - Declaration of functions
  2. linking function symbols referring to standard library functions
     - Definition is provided by the standard library implementation (i.e., C runtime)
     - Often implemented by mixture of assembler and C itself

Standard Library Modules (1)

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

Input and Output

- generic file operations for byte characters
- I/O streams associated with an external physical device (files, printer,...)
  - Standard input
  - Standard (error) output
- File access functions
  - opening & closing, aliasing, synchronization, buffer operations
- Direct input and output
  - reading from and writing to files
- Unformatted and formatted input and output
  - character/string reading from input stream, writing to output streams
- File positioning, error handling, file operations (remove, rename,...)

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (2)

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

Diagnostics

- runtime assertions (user-specified condition to abort program)
  - disable for release builds
  - *example*: check whether argument is null or not
- compile-time assertions (user-specified condition to stop compilation)
  - *example*: check whether implementation-defined type size in in certain boundaries

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (3)

```
#include <complex.h>
```

C99 or later

Complex Number Arithmetics

- complex number math
  - arithmetic operators for real, complex, and imaginary types in any combination
- manipulation functions
  - construction, computations of real, imaginary, magnitude, phase, ...
- exponential functions
  - complex base-e exponential and natural logarithm
- power functions
  - complex power and square root
- trigonometric functions
  - complex sine, cosine, tangent, arc sine, arc cosine, and arc tangent
- hyperbolic functions
  - hyperbolic sine, hyperbolic cosine, hyperbolic tangent, ...

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (4)

```
#include <ctype.h>
```

Null-Terminated Byte Strings Tests

- character classification functions
  - check for alphanumeric, alphabetic, lower/uppercase, digit, hexadecimal, control/ graphical character, spaces/blanks, printing & punctuation character
- character manipulation functions
  - conversion to lowercase and uppercase
- numeric formats conversion
  - byte string to a floating-point, integer, unsigned integer, and floating point
- string manipulation and examination
  - strings/substrings copying & concatenation, localization for string comparing
  - lengths, comparing (sub)strings, first/last occurrence search, string tokenizing
- character array manipulation
  - searching of elements in array, comparing byte arrays, copying & moving (sub) arrays
- printing error codes as human readable (string) messages

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (5)

```
#include <errno.h>
```

Error Reporting

- POSIX-compatible
  - thread-local error numbers
  - error conditions
- boundary checking

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (6)

```
#include <fenv.h>
```

C99 or later

Floating Point Environment

- thread-local set of status flags and control modes after floating-point operations
- status flags indicate abnormal results or auxiliary information
- native support only by few compilers
- functions
  - flag clearing, getting certain flag, exceptions raising (division by zero, inexact, invalid, overflow, underflow), copying flags, getting/setting round directions (downward, towards zero, nearest, upwards), saving/restoring current states,

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (7)

```
#include <float.h>
```

```
#include <limits.h>
```

Implementation-defined Limits of Float and Basic Types

- floating-point types
  - decimal digits, min, max, epsilon, mantissa-specific information, number of digits guaranteed preserved in text, . . .
- basic types
  - library types limits
    - pointer arithmetic, array indexing and loop counting types, atomic integers,. . .
  - integer types limits
    - number bits per byte, max number of bytes in multibyte character
    - min/max value of (signed/unsigned) character, short, int, long, and long long

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (8)

```
#include <inttypes.h>
```

C99 or later

Format Conversion of Integer Types

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (9)

```
#include <iso646.h>
```

C95 or later

Alternatives to Operator Spellings

C operators and punctuators require `{` , `}` , `[` , `]` , `#` , `\` , `^` , `|` , `~` characters that might not be available in a particular character encoding (e.g., DIN 66003)

Alternatives to those characters:

- macros
  - text substitution, e.g., `and` for `&&` , `and_eq` for `&=` ,...
- digraphs
  - substitute two characters by one, e.g., `<%` for `{` , `<:` for `[` , ...
- trigraphs
  - substitute three characters by one, e.g., `??<` for `{` , `??(` for `[` ,...

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (11)

```
#include <locale.h>
```

Localization Support

Localization is adapting a software to different regions, and different languages.
Adaption includes technical differences (e.g., character encoding).

Localization affects

- character classification and string comparison
- formatting of numbers, currency, and date/time
- behavior of stream I/O, regular expression
- ...

The standard library provides function to individually get and set localization incl. formatting details .

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (12)

```
#include <math.h>
```

Common Mathematical Functions

- basic functions
  - absolute value, quotient and remainder, division, min/max, differences for ints/floats
- exponential and power functions
  - base-2, base-10, and base-e exponential, base-2, base-10m and natural logarithm
  - power, square root, cubic root, hypotenuse calculation
- trigonometric and hyperbolic functions
  - complex sine, cosine, tangent, arc sine, arc cosine, and arc tangent
  - hyperbolic sine, hyperbolic cosine, hyperbolic tangent, ...
- error and gamma functions
  - statistical error, complementary, gamma, and natural logarithm of gamma function
- nearest integer floating-point functions
  - ceil, floor, trunc, rounding with several modes
- floating-point manipulation, float classification and comparison, macros, constants,...

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (13)

```
#include <setjmp.h>
```

Non-local Jumps

Unconditional control flow jump (c.f. `goto` *later*), specialized for jumps between functions.

`jmp_buf` environment and context

- buffer for restoring execution environment, and for invocation of that environment

`setjmp` non-local jumps

- save current context in a buffer used to restore execution context later on

`longjmp` context execution

- loading of execution environment, and transferring control flow to jump target

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (14)

```
#include <signal.h>
```

Signal Handling

Signals is POSIX-compliant form of inter-process communication in terms of asynchronous notifications to processes (or threads) on the happening of an event, such as

- program termination (`SIGABRT`), interruption (`SIGINT`), or killing (`SIGKILL`)
- elapsing of particular timers (`SIGALRM`)
- job control events, e.g., pausing (`SIGSTOP`) or continuing (`SIGCONT`)
- errors, e.g., bus error (`SIGBUS`), or division-by-zero (`SIGFPE`)
- ...

Functions and macros for signal handling in C programs

- setting signal handler for particular signals
- invocation of signals for the current program

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (15)

```
#include <stdalign.h>
```

C11 or later

Alignment Requirements and Alignment Information

Alignment requirement of a type represents the number of bytes (power of two) between successive addresses at which objects of this type can be allocated.

Support of modification and querying of per-type alignment requirements, e.g., of members in structure and unions.

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (16)

```
#include <stdarg.h>
```

Variable Number of Arguments for Functions

Support for functions taking a variable number of arguments (aka variadic functions), e.g., `printf`

declaration: `int foo(int argument, ...);`
usage: `foo(42, 23, „Hello“, „World“);`

Support in function definition enabled by specialized macros to

- enable variadic arguments
- access next variadic argument
- copying variadic arguments
- ends traversal of variadic argument list

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (17)

```
#include <stdatomic.h>
```

C11 or later

Atomic Types and Operations

Requirements and semantics of multi-threaded programs to enable portable multi-threaded programs efficiently manipulating objects in parallel without data races.

- memory order constraints
  - ordering of non-atomic memory accesses around an atomic operation
    - sequentially-consistent ordering, relaxed ordering, release-consume ordering,...
- atomic functions
  - atomic write, read and (conditional) swap operations, atomic test and set and clear, check for lock-freeness, atomic addition, subtraction, logical OR, logical exclusive OR, logical AND, synchronization primitives
- atomic types
  - atomic types for core languages types, guaranteed lock-free atomic flag

taken from: http://en.cppreference.com/w/c/header

Standard Library Modules (18)

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

C99 or later

Boolean Type Support

Older C language specifications don't support dedicated boolean types (i.e., int was used)

Since C99, a boolean type `bool` (a macro expanded to built-in type `_Bool`) can be used

- application of standard logical operators in any combination
- provisioning of two macro constants `true` (integer `1`) and `false` (integer `0`)

taken from: http://en.cppreference.com/w/c/header