Leibniz Universität Hannover

Programmieren 1

C Introduction

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations and intervals	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
online → 11	23.12.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	13.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	20.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	
14	27.1.	C language (remaining C keywords), finite state machines, quicksort	(14)	

Review

- Key-value arrays

```
[x: 10 y: 20]
```

- Data definitions
- Compound Data (Product Types)

```
Point: (x :Num, y :Num) datadef
```

- Variant Data (Sum Types)

```
Point: { Euclid: (x :Num y :Num),
         Polar: (theta :Num, mag: Num)
} datadef
```

- Recursion
- Self-Referential Data (Recursive Types)

Preview

- Execution model of C
- Variables and constants: declaration, definition
- Conditional execution: if statement
- Functions: declaration, definition
- Programming I C library
- Atomic Data (in C)
- Loops: while, for, do-while

C INTRODUCTION

The C Programming Language

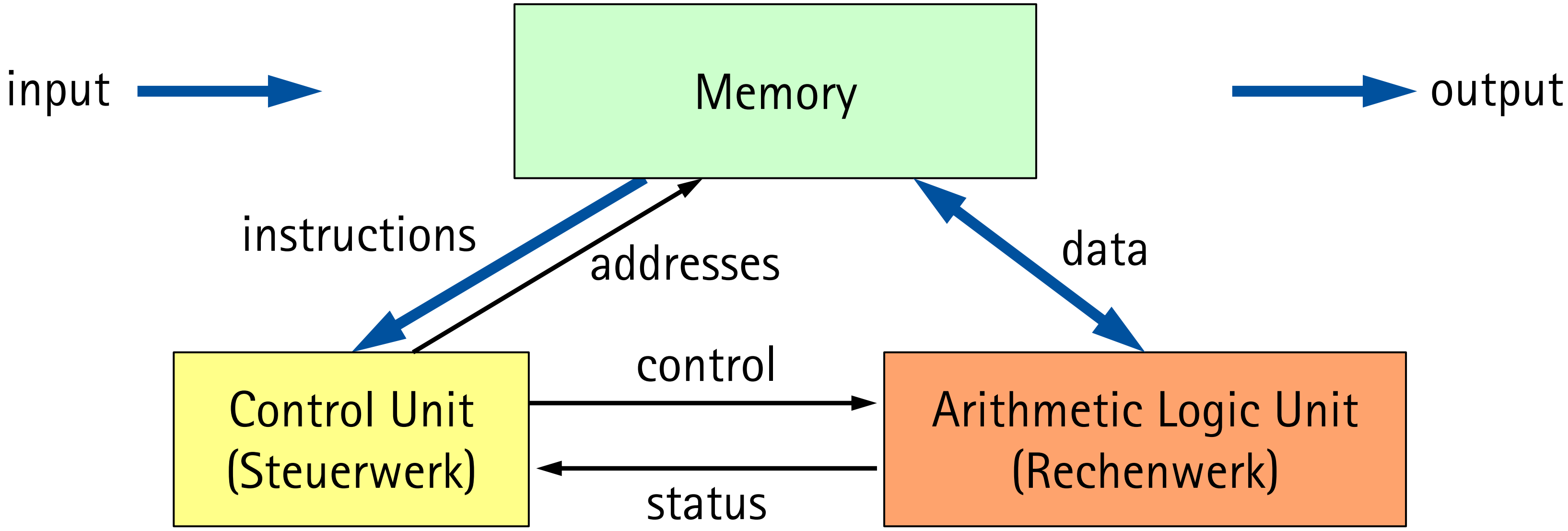
- One of the most widely used programming languages of all time
  - Influenced many later languages, for example, Java
- Developed by Dennis Ritchie at Bell Labs in 1972
  - Designed as a system programming language for UNIX
  - Predecessor: BCPL, B (both typeless)
- Features
  - Syntactically small, few keywords
  - Allows machine-oriented programming
  - Supports structured and modular programming
- History
  - 1983 ANSI working group begins standardizing C
  - 1989 ANSI publishes “ANSI C” / ISO “C89” standard
  - 2018 ISO Standard C17 is current standard

C Keywords (ANSI C / ISO C89)

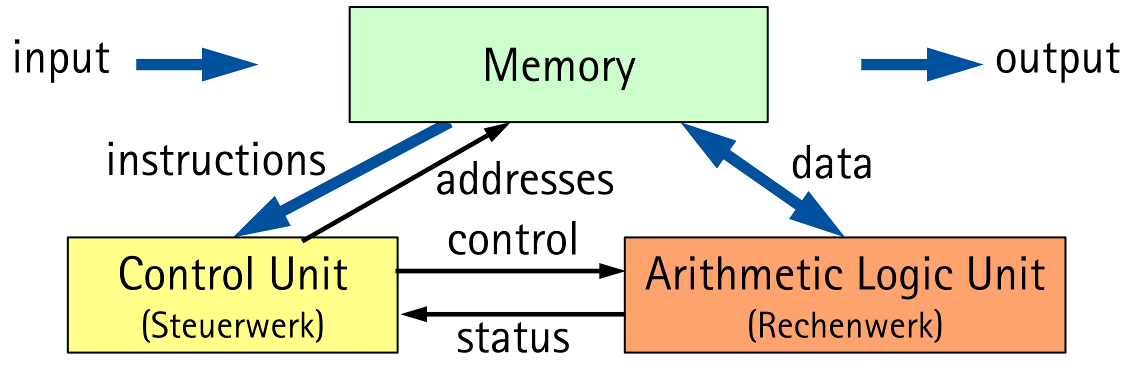
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Execution Model: Von Neumann Architecture

- von Neuman architecture:
  data and instructions stored in memory
- Typical architecture to which C source code is translated



Execution Model: Von Neumann Architecture

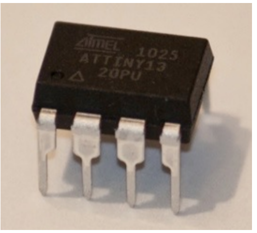


Computer repeats this forever

1. Fetch instruction (memory → control unit)
2. Decode instruction (in control unit)
3. Load data (memory → arithmetic logic unit, ALU)
4. Compute result and status bits (in ALU)
5. Store result (ALU → memory or ALU register)
6. Compute memory address of next instruction

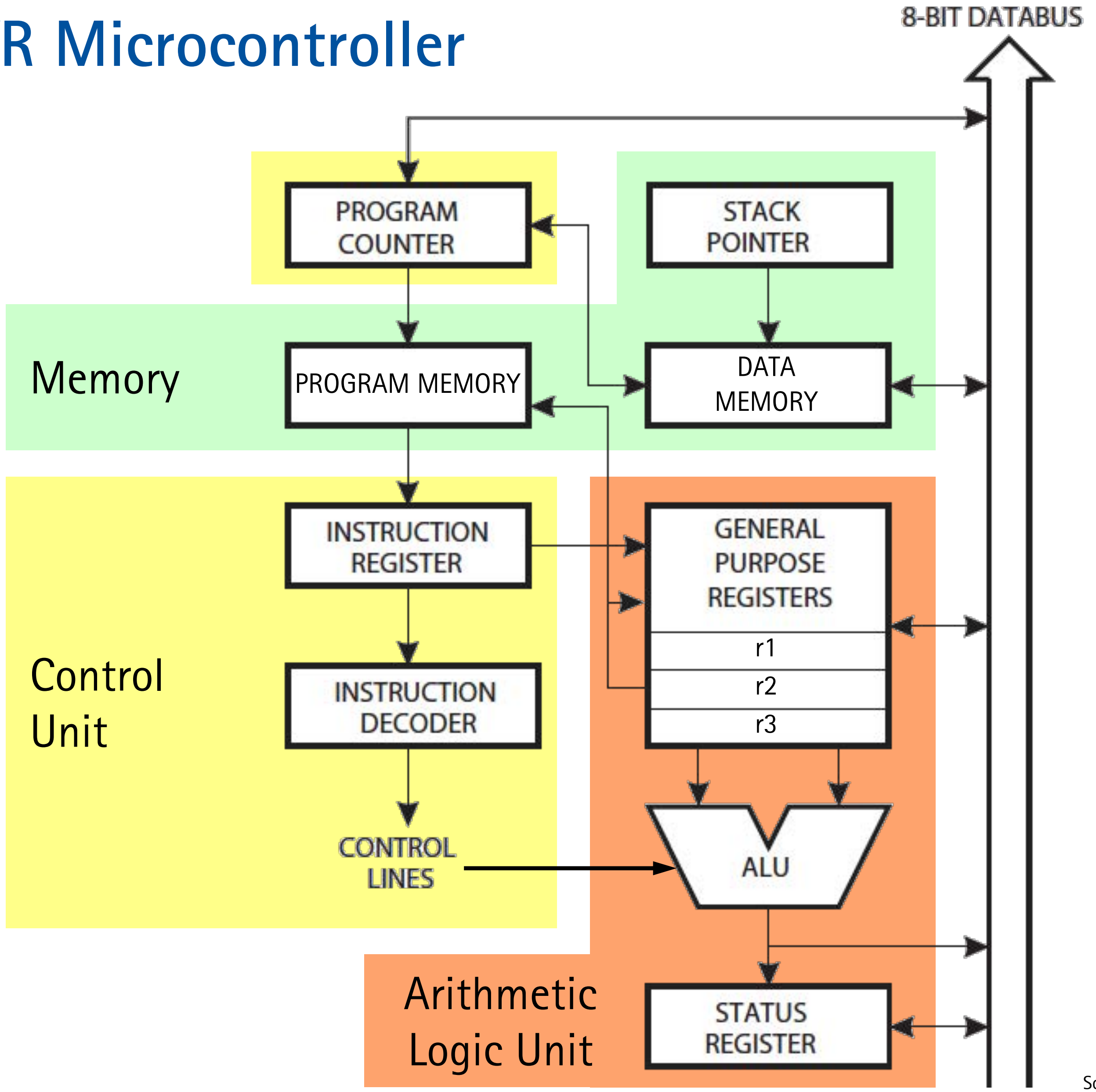
Machine Langue vs. High-Level Language

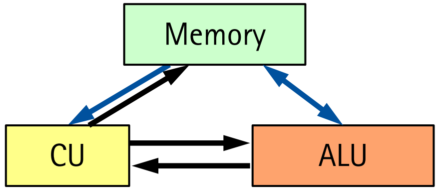
- Machine langue depends on the microprocessor architecture
- High-level languages abstract from these details

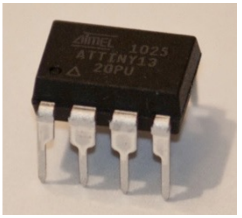


Machine language (here: AVR microcontroller)		High-level language
lds r1, 1000	// load cell 1000 (x)	z = x + y;
lds r2, 1001	// load cell 1001 (y)	
add r1, r2	// add r1 and r2 and store in r1	
sts 1002, r1	// store result in cell 1002 (z)	

Example: AVR Microcontroller







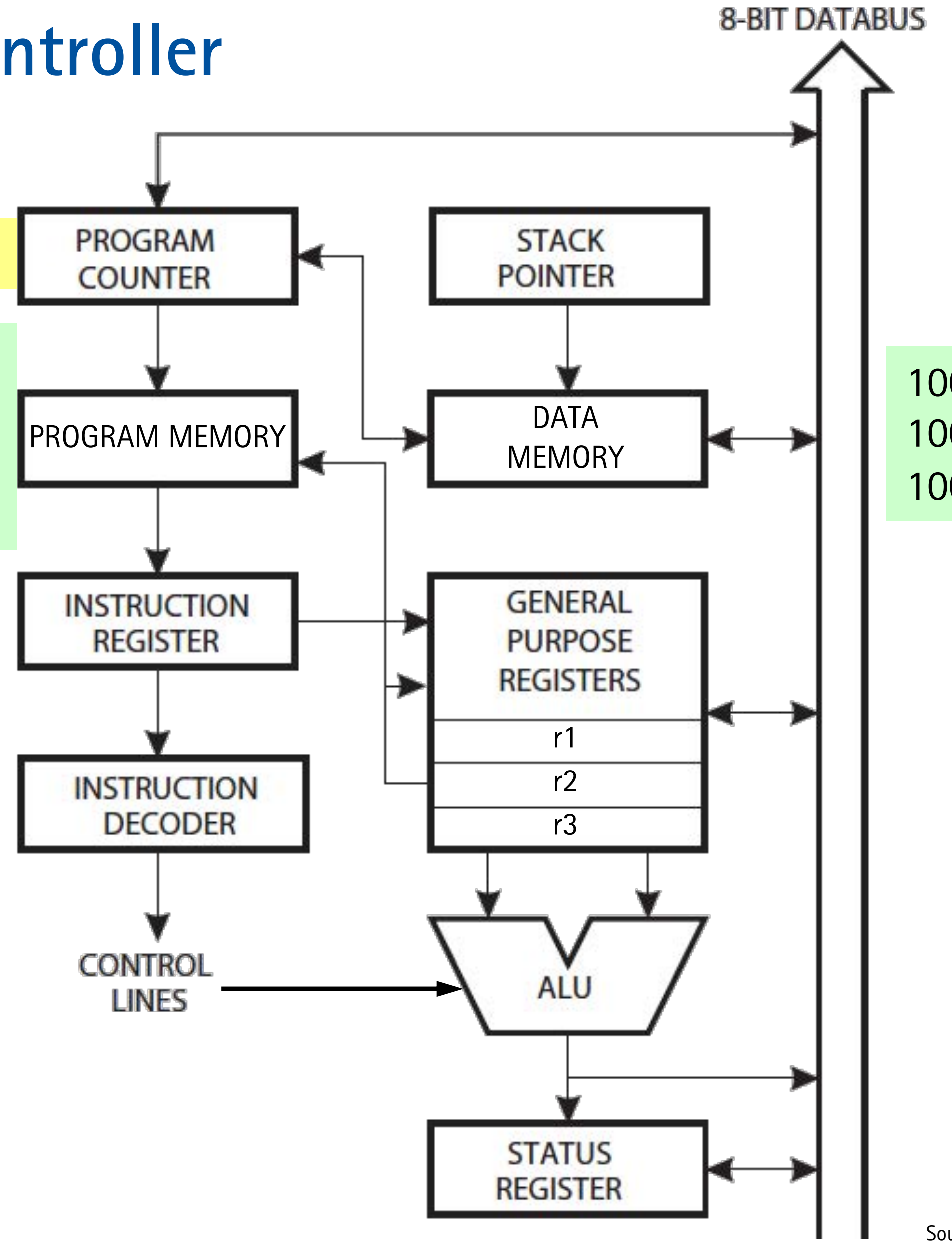
ATtiny45

Source: AVR ATtiny45 datasheet

Example: AVR Microcontroller

0

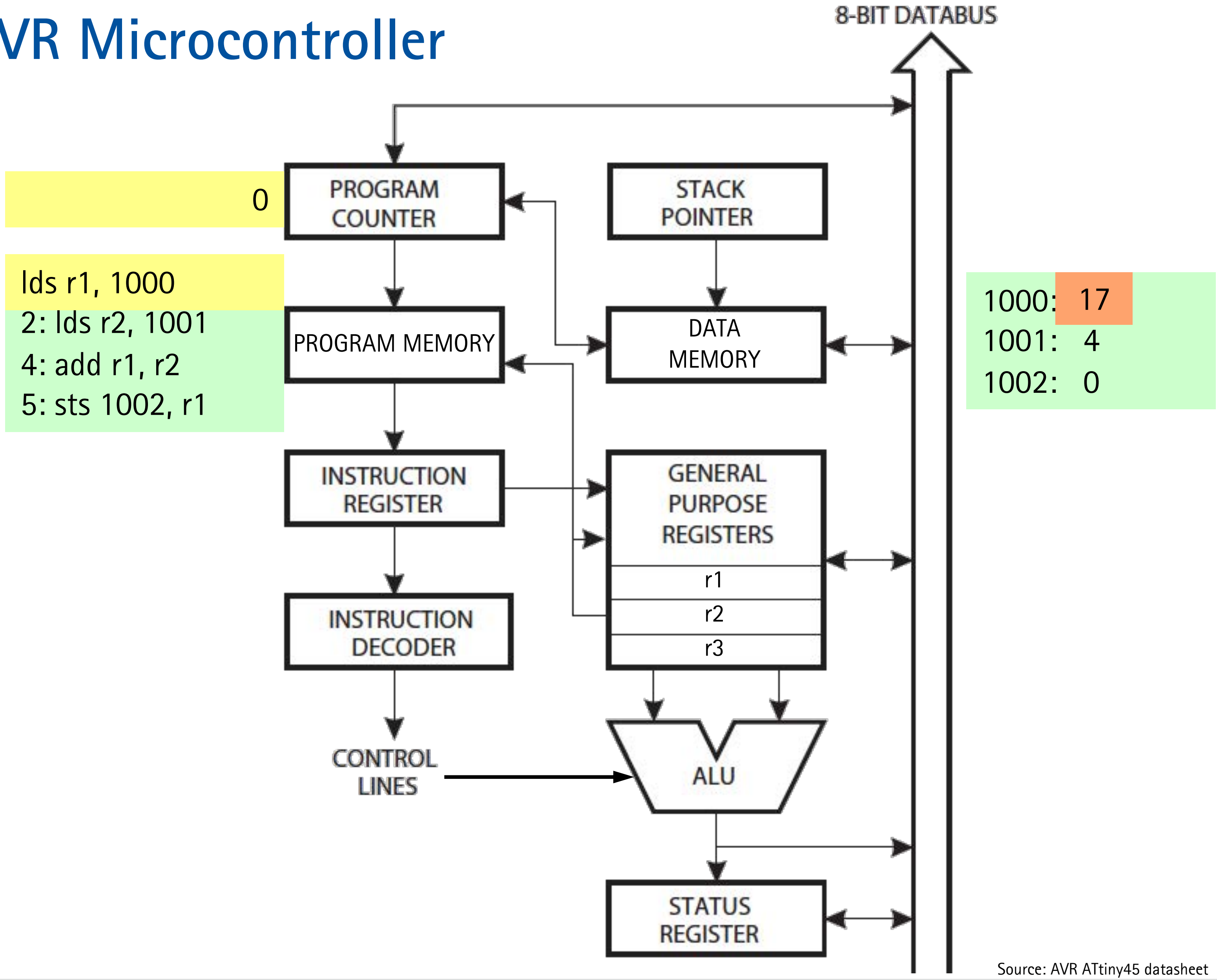
0: lds r1, 1000
2: lds r2, 1001
4: add r1, r2
5: sts 1002, r1



1000: 17
1001: 4
1002: 0

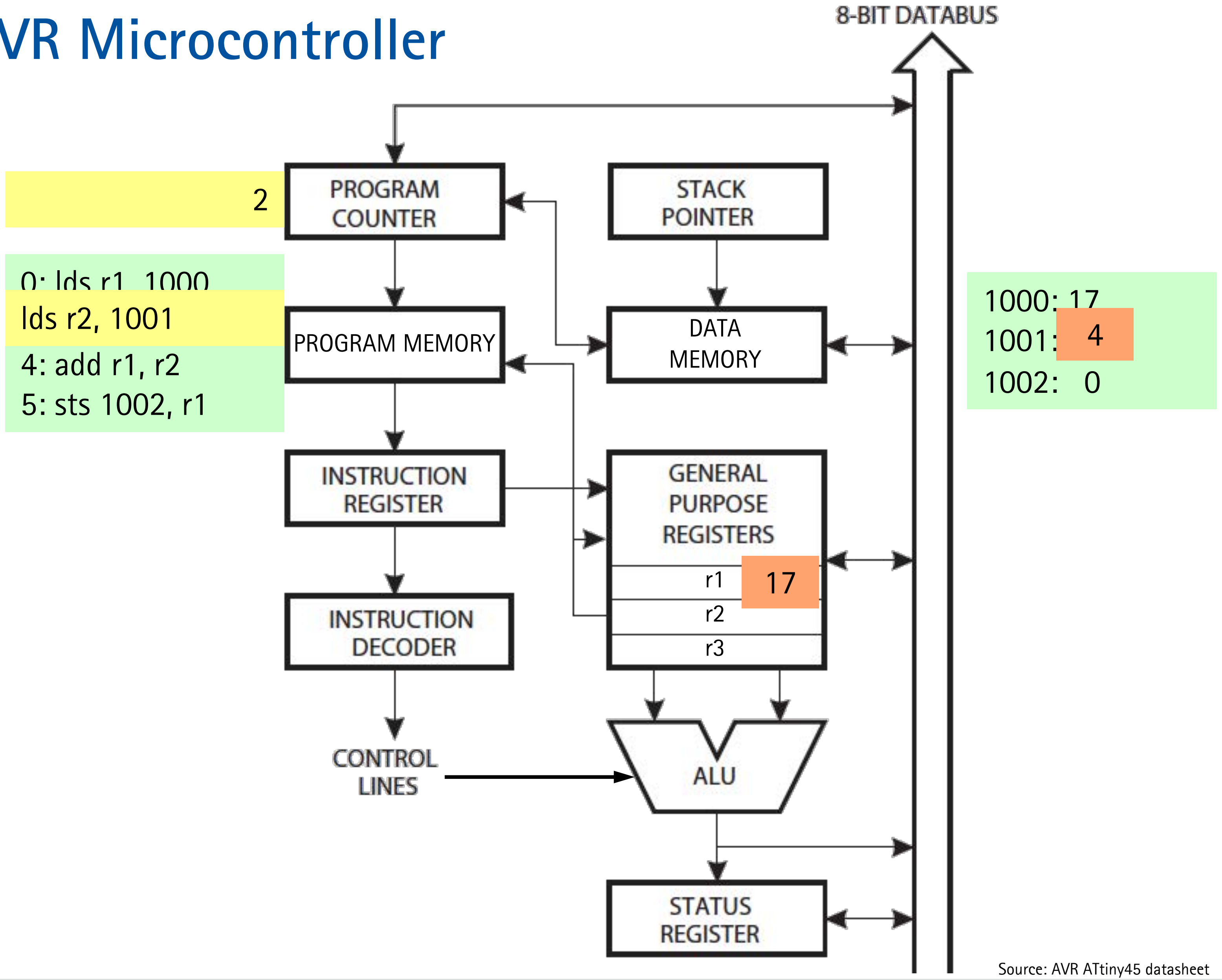
Source: AVR ATtiny45 datasheet

Example: AVR Microcontroller



Source: AVR ATtiny45 datasheet

Example: AVR Microcontroller

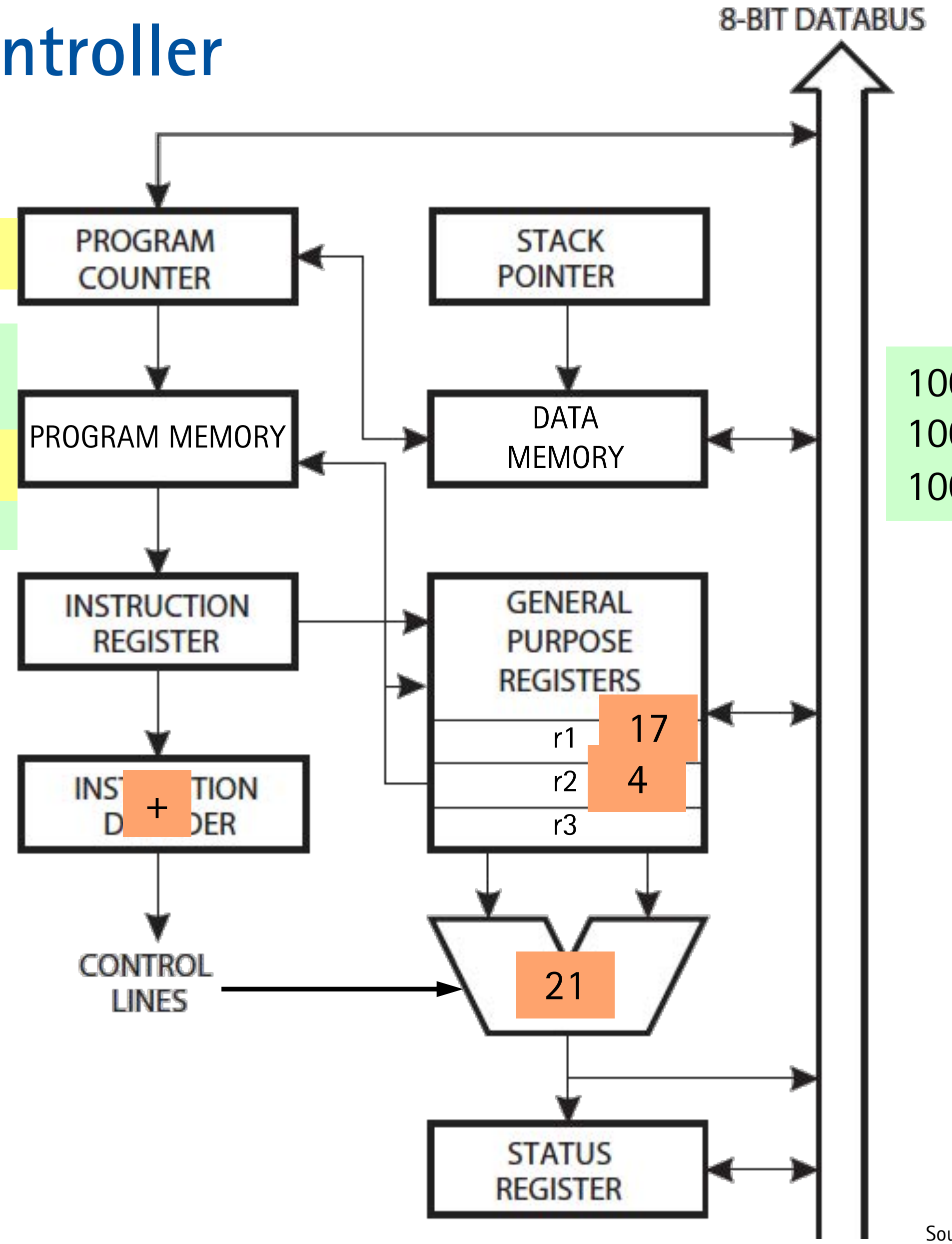


Source: AVR ATtiny45 datasheet

Example: AVR Microcontroller

4

0: lds r1, 1000
2: lds r2, 1001
add r1, r2
5: sts 1002, r1



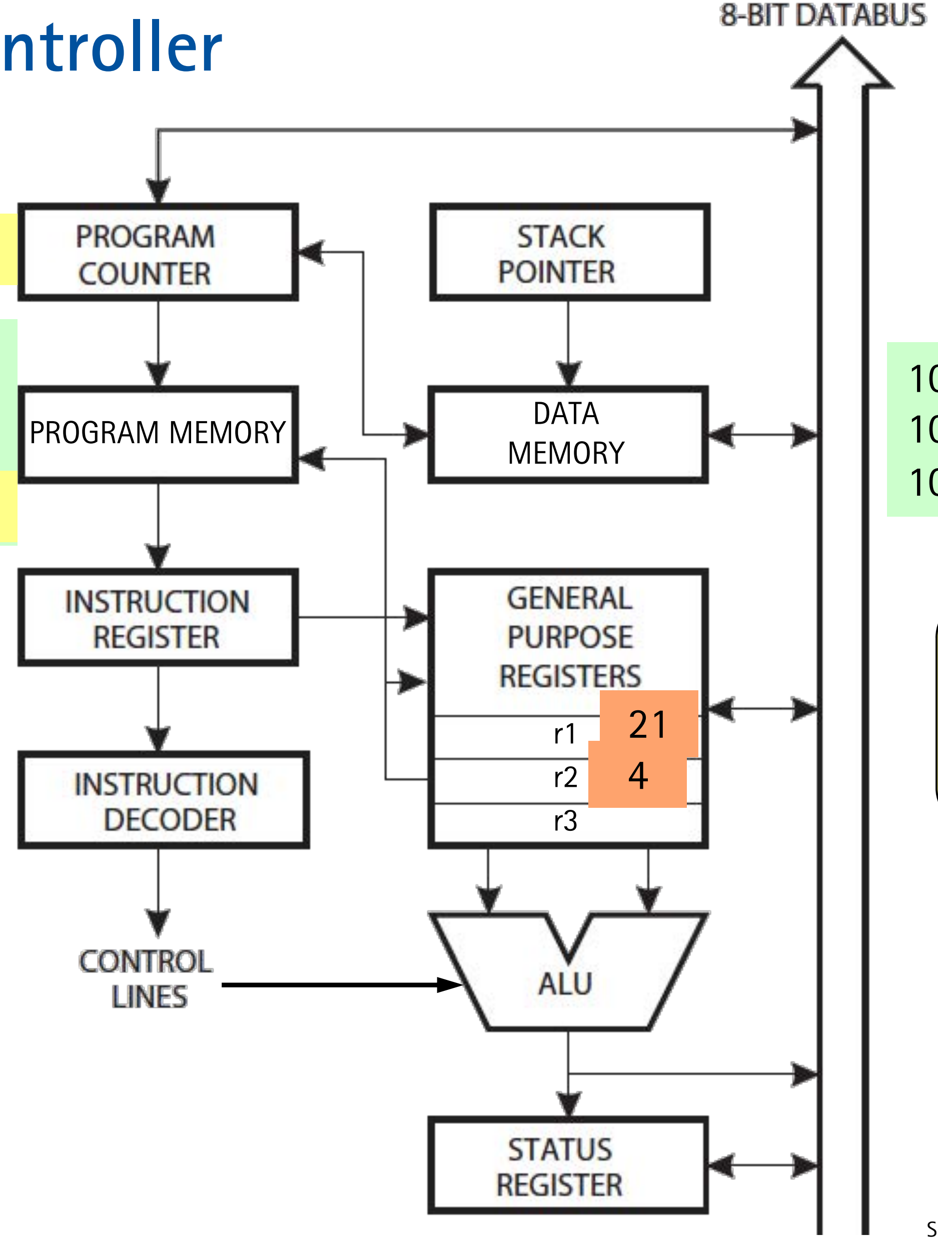
1000: 17
1001: 4
1002: 0

Source: AVR ATtiny45 datasheet

Example: AVR Microcontroller

5

0: lds r1, 1000
2: lds r2, 1001
4: add r1, r2
sts 1002, r1



1000: 17
1001: 4
1002: 21

Instructions: Operations on memory cells

Source: AVR ATtiny45 datasheet

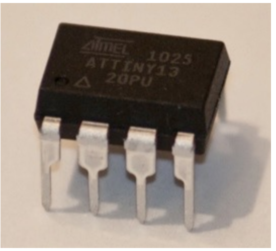
Machine Langue vs. High-Level Language

High-level language

```
z = x + y;
```

Machine Langue vs. High-Level Language

High-level language	Machine language
	lds r1, 1000
z = x + y;	lds r2, 1001
	add r1, r2
	sts 1002, r1

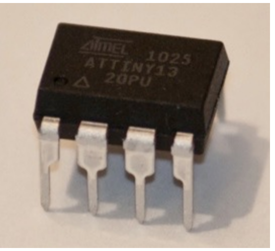


Atmel AVR ATTiny13

Machine Langue vs. High-Level Language

High-level language	Machine language	Machine code	
	lds r1, 1000	1 0 0 1 0 0|0|00001|0 0 0 0	1000_{10} (16-bit address)
z = x + y;	lds r2, 1001	1 0 0 1 0 0|0|00010|0 0 0 0	1001_{10} (16-bit address)
	add r1, r2	0 0 0|0|1 1|0|00001| 0010	
	sts 1002, r1	1 0 0 1 0 0|1|00001|0 0 0 0	1002_{10} (16-bit address)

1	**0**	**0**	**1**	**0**	**0**	**s**	**d d d d d**	**opcode**				**Load/store operations**
1	0	0	1	0	0	s	d d d d d	0	0	0	0	LDS rd,i/STS i,rd
16-Bit immediate SRAM address i												
0	**0**	**opcode**				**r**	**d d d d d**	**r r r r**				**2-operand instructions**
0	0	0	cy	1	1	r	d d d d d	r r r r				ADD/ADC Rd,Rr (LSL/ROL Rd when Rd=Rr)



Atmel AVR ATTiny13

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

Machine Language vs. High-Level Language

Machine language (here: AVR microcontroller)		High-level language
lds r1, 1000	// load cell 1000 (x)	z = x + y;
lds r2, 1001	// load cell 1001 (y)	
add r1, r2	// add r1 and r2 and store in r1	
sts 1002, r1	// store result in cell 1002 (z)	

- Memorizing these steps is tedious
- Programmer should focus on problem, not hardware internals
- High-level programming languages
  - abstract from hardware details
  - are closer to how we think as humans

Hello World in C

- <irony>Law #1: The only way to learn a programming language is to write a program that prints the words: hello, world</irony>

- In C:

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

include information from standard library (printf)

define function named “main” that takes no parameters and returns an integer, predefined entry point to any C program

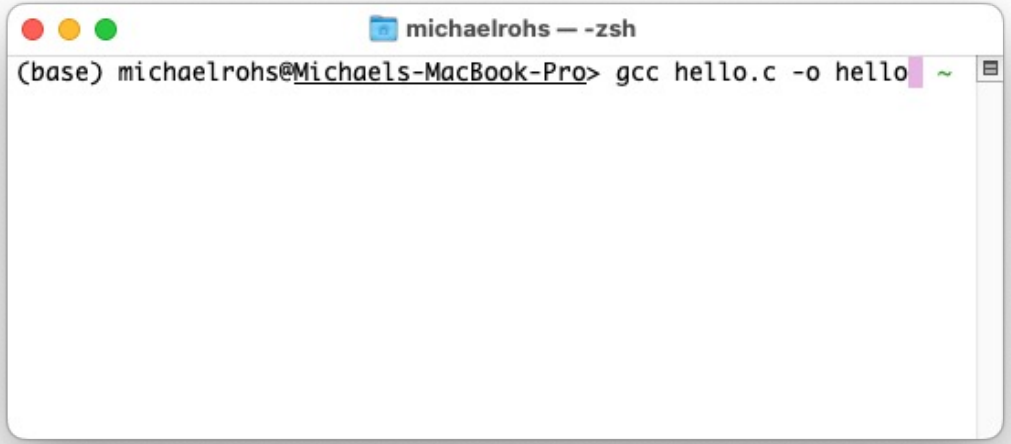
statements of main are enclosed in braces

main calls library function printf to print sequence of characters, \n represents the newline character

return value 0 to indicate success

hello.c

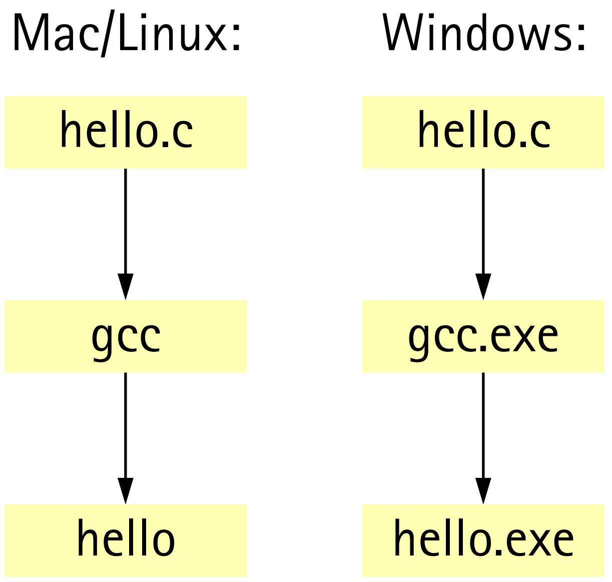
Compiling and Running Hello World in C



- Enter source code in text editor
- Save as hello.c
- Open terminal / command line
- Change to directory containing hello.c
- Compile with GNU C Compiler (GCC):
  Mac/Linux: `gcc hello.c -o hello` Windows: `gcc.exe hello.c -o hello.exe`
  input: source file
  output: executable file
- Run from command line:
  `./hello`
  "." refers to current directory
- Output:
  hello, world

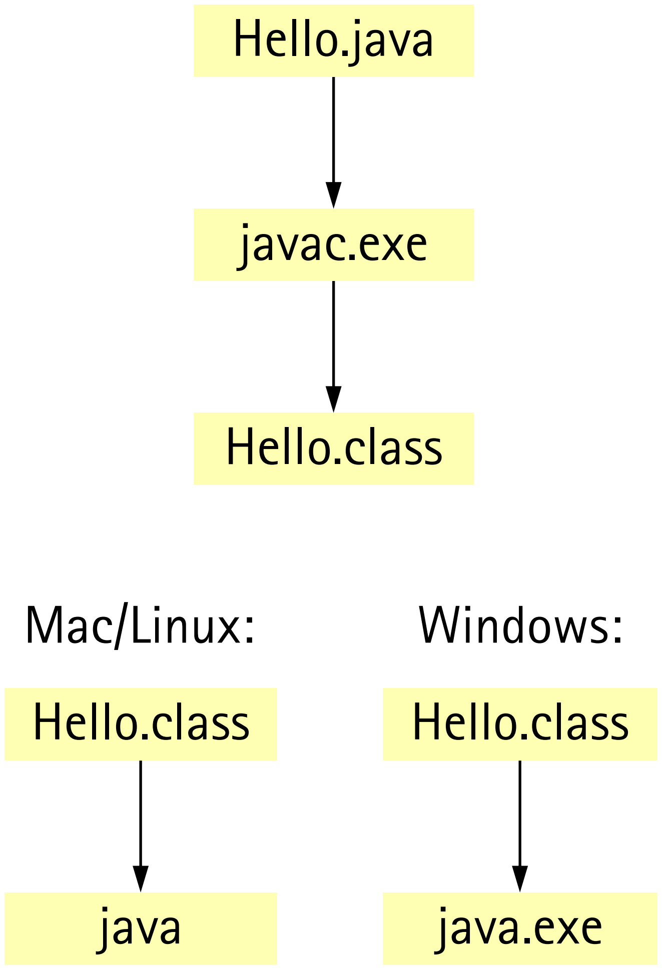
Interpreter vs. Compiler

- **Interpreter:** A program that reads a source file and executes the instructions it contains
  - Example: python hello.py
  - Platform-independent (typically)
- **Compiler:** A program that translates a source file to machine code and creates an executable file for a particular platform
  - Example: gcc hello.c -o hello.exe
  - Compile time: read hello.c, produce hello.exe
  - Run time: execute hello.exe
  - Platform-specific (processor, operating system)



Bytecode Compiler and Interpreter

- Bytecode compiler produces bytecode from source code
  - Example: javac Hello.java
  - Produces Hello.class
  - Platform-independent
- Bytecode interpreter executes bytecode
  - Example: java Hello
  - Executes Hello.class
  - Stack-oriented virtual machine
  - More efficient to interpret than interpreting source code
  - No syntax errors at this stage



celsius_to_fahrenheit.c

```c
#include "base.h"
// Takes a temperature value in degrees Celsius and
// returns the corresponding value in degrees Fahrenheit.
int celsius_to_fahrenheit(int celsius) {
   return celsius * 9 / 5 + 32;
}
int main(void) {
   printiln(celsius_to_fahrenheit(0)); // given 0, expect 32
   printiln(celsius_to_fahrenheit(10)); // given 10, expect 50
   printiln(celsius_to_fahrenheit(-5)); // given -5, expect 23
   printiln(celsius_to_fahrenheit(100)); // given 100, expect 212
   return 0;
}
```

include file base.h from the Programming I library (details later)

main function: entry point or program, takes no arguments (void), returns an integer number (int)

printiln: print an integer followed by a line break

return 0 to indicate success

Compiling the Program (on the command line)

> make celsius_to_fahrenheit

the programmer types on the command line

gcc -std=c99 … -o celsius_to_fahrenheit

the compiler reads celsius_to_fahrenheit.c and produces celsius_to_fahrenheit.exe

script_examples — -bash — ⌘1

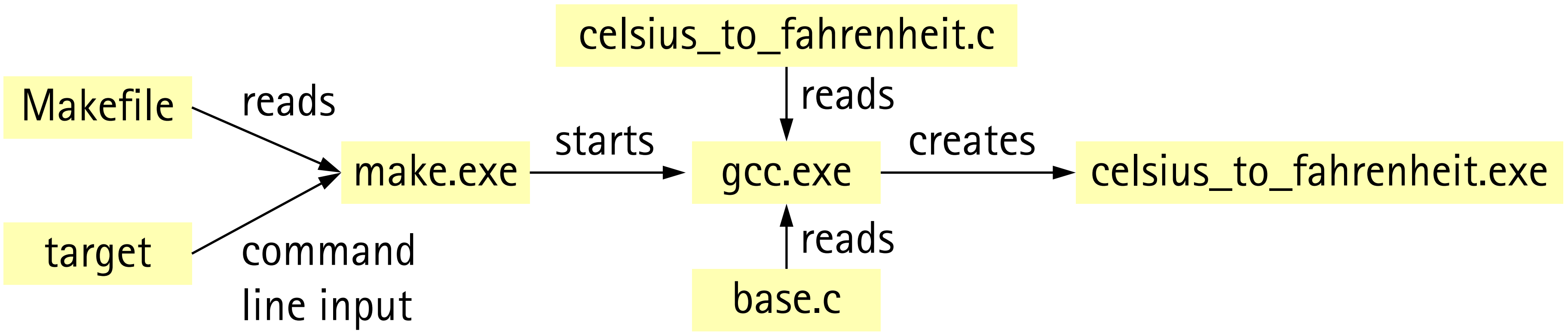
```
MacBook-Pro-2:script_examples michaelrohs$
MacBook-Pro-2:script_examples michaelrohs$ make celsius_to_fahrenheit
gcc -std=c99 -Wall -Werror -Wpointer-arith -Wfatal-errors -g celsius_to_fahrenheit.c -L../lib -lprog1 -iquote../lib -o celsius_to_fahrenheit
MacBook-Pro-2:script_examples michaelrohs$
```

Compiling the Program (on the command line)

- The C source code is stored in the text file celsius_to_fahrenheit.c
- The GNU C compiler (gcc) transforms the C source code into an executable program (for Windows, for Linux, for Mac, etc.)
- The make tool to simplifies the build process
  - make is a program that reads a Makefile in the current directory
  - the Makefile has rules that tell make how to call gcc to produce the target (here the target is celsius_to_fahrenheit)

> make celsius_to_fahrenheit

target



Preprocessor, #include

- Lines beginning with #<keyword> are preprocessor directives
  - Preprocessor runs before compiler
  - Preprocessor performs textual replacements in the source code

- #include directive
  - Preprocessor replaces line #include “file.h” with contents of file.h
  - Used, e.g., to make function headers available
  - Example: #include “base.h”, contains, e.g., header void println(int i);

Error Messages are your Friend

- C compiler sometimes reports errors 😠
- Try to interpret these error messages
- They are often helpful
- Extract at least the line and column number
- Example

```
ggt.c:22:19: fatal error: too few arguments to function call, expected 2, have 1
        else return ggt(y);
                    ~~~  ^
ggt.c:20:1: note: 'ggt' declared here
int ggt(int x, int y) {
```

- Sometimes the cause is far away
  - example: missing opening or closing braces

Basic Elements of a C Program

- Names
  - consist of letters, digits, underscore _
  - begin with letters or underscore
  - lower/upper case is significant
- Keywords
  - have a special meaning
  - cannot be used as names
- Numbers
  - integer numbers (decimal, hexadecimal)
  - floating point numbers
- Strings
  - arbitrary characters between double quotes
  - may not cross lines

```
celsius_to_fahrenheit
println
my_var
```

```
if
return
```

345	decimal
0x23a	$2*16^2+3*16^1+10*16^0$
3.14	floating-point

```
"a simple string"
"she said 'hello'"
```

Basic Elements of a C Program

- Operators
  - Denote operations on operands

+, -, *, /, !, =, ==, ->, ., etc.

- Brackets
  - Group elements to a larger unit

Brackets [ ]
Parentheses ( )
Braces { }

- Semicolon
  - Terminates statements and declarations
  - Line break does not terminate a statement
  - Could write a complete C program in a single line

;

From Basic Elements to Larger Syntactic Units

- Compose basic elements to form larger syntactic units
- Expressions
  - Compose expressions from operands (x, y) and operators (+)
  - Expressions reduce to a value

```
x + y
```

- Statements
  - Compose statements with semicolon (;)
  - A single action of the program
  - A single step of computation

```
z = x + y;
```

- Blocks
  - A sequence of statements that are treated as a unit

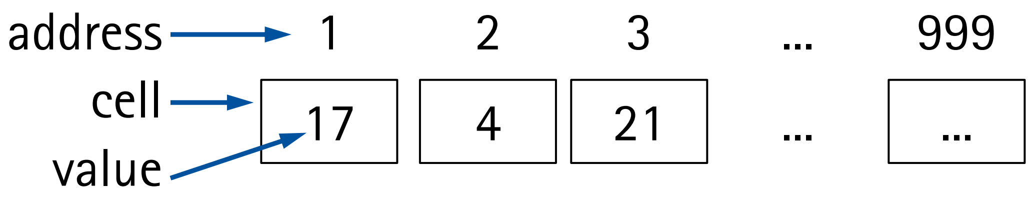
```
{ z = x + y;
  a = b * c;
  d = d + 1; }
```

- Functions
  - An algorithm, a named piece of computation
  - Function header and implementation

```
int f (int x) {
  return 2 * x; }
```

Program = Data and Instructions

- **Data:** A set of addressable memory cells

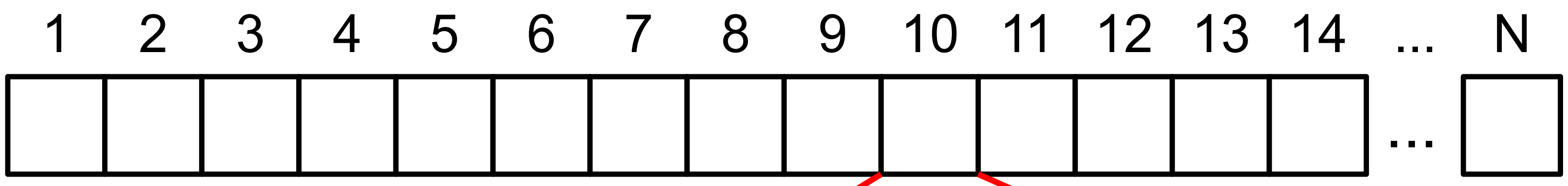


  - Data are stored as binary values (e.g., $5 = 101_2$)
  - Binary values are universal (numbers, texts, images, sounds, etc.)
  - Binary values have to be interpreted correctly

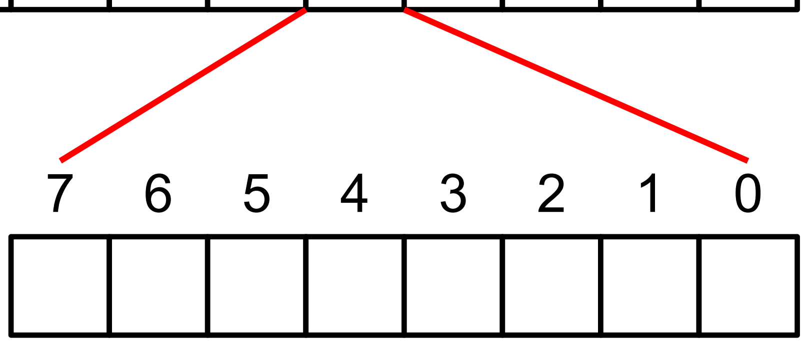
- **Instructions:** Operations on memory cells

Memory in C

- A linear sequence of bytes, numbered from 1 to N



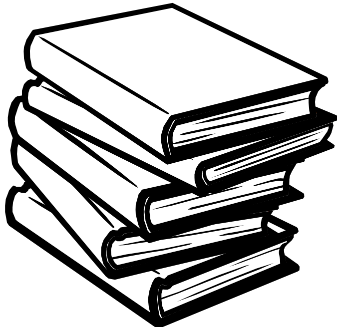
- Each byte contains a bit pattern (8 bits)



- C allows free access to memory cells
- No strict abstraction barriers as in other languages

Memory in C

- Within that memory there is a stack (as in PostFix)

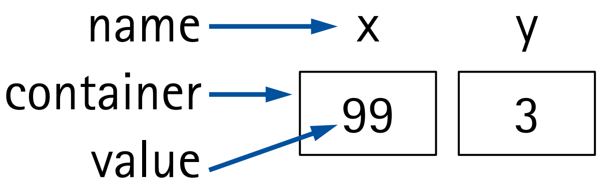


- and a heap

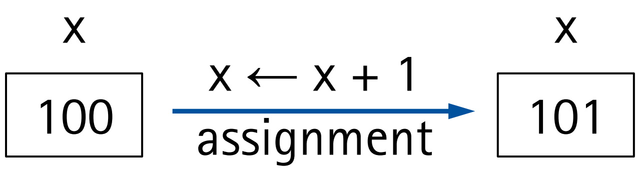
- but no dictionary (different from PostFix)

Variables

- Named containers for values



- Can change their value



- Have a data type = set of allowed values

data type of variable	values of variable	type ≅ form
number	17, 54, …	a number variable can only store numbers
character	'a', 'x', …	a character variable can only store characters

Variable Declarations

- In C variables need to be declared before use
  - Makes the name and type of the variable known to the compiler
  - Compiler reserves memory for the variable's content
- Built-in types
  - int x;   declares a variable x of type integer (often 32 bits)
  - char c;   declares a variable c of type character (8 bits)
  - double d; declares a variable d of type double-precision floating-point number (64 bits)
  - other types later...
- Style
  - Use lower-case variable names
  - Use underscore as a word separator, e.g., hours_to_wages

Variable Declarations

- Initial value undefined, if not explicitly given
  - `int i;`
- Variable can be initialized in declaration
  - `int i = 0;`
  - `char newline = '\n';`
  - `double eps = 1.0e-5;`

In C "=" means assignment, not equality!

In C "==" tests for equality

Constants

- Constants behave like variables whose values cannot be changed after initialization
  - "read-only variables"
- Example: `const int WEEKLY_HOURS = 40;`
  - Better readability, "WEEKLY_HOURS" says more than "40"
- Better maintenance
  - Changing the value of the constant in one place only
  - Don't repeat yourself!
- Style
  - Use CAPITALIZED_WITH_UNDERSCORES for constants

Conditional Execution in C: if-Statement

- if-Statement:

```
if (condition) {
    condition-true-part
} else {
    condition-false-part
}
```

- The else-part is optional

```
if (condition) {
    condition-true-part
}
```

- Semantics:
  - **condition** is evaluated
  - If **condition** is true, then the condition-true-part is executed
  - Otherwise the condition-false-part is executed

- Parts may be single statements instead of blocks

```
if (condition) statement;
```

Chained if-Statements

- else-part may be another if-statement

```
if (condition₁) {
    "condition₁-true-part"
} else {
    "condition₁-false-part"
}
```

- replace else-part by a second if statement:

```
if (condition₁) {
    "condition₁-true-part"
} else if (condition₂) {
    "condition₂-true-part"
} else {
    "condition₂-false-part"
}
```

Nested if-Statements and the Dangling else

- The else part goes with the inner if

```
if (n > 0)
if (a > b) z = a;
else z = b;
```

```
if (n > 0) {
    if (a > b) {
        z = a;
    } else {
        z = b;
    }
}
```

- The C compiler does not care about indentation
- Use blocks { … } to clarify!

Comments

```
/* This is a comment. */
```

- Explain, give additional information, no runtime effect
- Single-line comments
  - Start with //
  - Go to the end of the line

```
int sum; // total sales
```

- Multi-line comments
  - Enclosed with /* ... */
  - Can run over multiple lines
  - Can be used to comment out parts of a program

```
/* print Fahrenheit-Celsius
   table for fahrenheit = 0,
   20, ..., 300 */
```

- Reasonable comments!
  - Comment what needs explanation
  - Clear source code is better than clear comments
  - Do not comment what is clear from the source code

```
int sum; // Summe
```

not helpful

FUNCTIONS

Function Definition

- Function definition

```
int hours_to_wages(int hours) {
    if (hours <= 40) {
        return hours * 10;
    } else {
        return 40 * 10 + (hours - 40) * 15;
    }
}
```

- Function call

```
hours_to_wages(45);
```

Function Declaration and Definition in C

- Function declaration (function header)
  - Describes the interface of the function
  - Descriptive names are important

```
int hours_to_wages(int hours);
```

- Function definition (function header + implementation)
  - Defines how the function computes its result

```
int hours_to_wages(int hours)
{
    ...
}
```

- Place function declaration or definition in .c file before using (calling) a function

- Calling a function

```
hours_to_wages(45);
```

Function Parameter(s) and Result have Types

- Type describes the kind of data
  - Integer number, text, currency, URL, 3D point, GPS coordinates, etc.
- Example: int -> int
  - Input (parameter type): an integer number (representing hours)
  - Output (result type): an integer number (representing Euros)

```
int hours_to_wages(int hours) {
    if (hours <= 40) {
        return hours * 10;
    } else {
        return 40 * 10 + (hours - 40) * 15;
    }
}
```

Functions Arguments and Local Variables

- Function arguments are passed “by value”
  - Called function receives argument values
  - No access to the variables of the calling function
  - Parameters are identical to initialized local variables
- Functions have local variables
  - Come into existence when the function is executed
  - Disappear when function execution finishes
  - Are only accessible within the function
  - Are not automatically initialized

```
int hours_to_wages(int hours) {
    int week_hours = 40;
    if (hours <= week_hours) {
        return hours * 10;
    } else {
        return week_hours * 10 +
               (hours - week_hours) * 15;
    }
}
```

Returning Results from Functions

- `return` <expression>;
  - evaluates expression and returns value to caller
  - exits the function
- `return;`
  - exits the function but does not return a value (void functions)
- A function with return "type" **void** does not need a return statement
  - **void** means "nothing"

```
int hours_to_wages(int hours) {
    if (hours <= 40) {
        return hours * 10;
    } else {
        return 40 * 10 + (hours - 40) * 15;
    }
}
```

```
void greeting(void) {
    prinsln("Hello!");
}
```

PROGRAMMING I C LIBRARY

Programming I C Library

- Location
  - https://postfix.hci.uni-hannover.de/files/prog1lib/index.html
  - prog1lib-1.4.2.zip
- Motivation
  - Simplify C programming for beginners
  - Allow writing interesting programs without needing to know some of the technicalities of C
  - Support the "design recipes" approach
- Downside
  - You have to be aware what is part of the C language, the Programming I C library, and the C standard library

Output
(Programming I C Library)

- Print an integer constant
```
printi(123);
```
- Print an integer variable x
```
printi(x);
```
- ...then print a line break
```
printiln(123);
```
- Print a double-precision floating point constant
```
printd(3.141592654);
```
- Print a string
```
prints("hello world");
```

#include "base.h"

void	**printi** (int i) Print an integer. More...
void	**printiln** (int i) Print an integer followed by a line break. More...
void	**printd** (double d) Print a double. More...
void	**printdln** (double d) Print a double followed by a line break. More...
void	**printc** (char c) Print a character. More...
void	**printcln** (char c) Print a character followed by a line break. More...
void	**prints** (**String** s) Print a String. More...
void	**printsln** (**String** s) Print a String followed by a line break. More...
void	**printb** (bool b) Print a boolean value. More...
void	**printbln** (bool b) Print a boolean value followed by a line break. More...
void	**println** () Print a line break. More...

Output: Complete Program
(Programming I C Library)

```c
// Compile: make output
// Run: ./output
#include "base.h"   // we use printi, printiln, printdln, printsln

int main(void) {
   printi(123);
   printiln(456);
   printdln(3.141592654);
   printsln("hello world");
   return 0;
}
```

lecture_examples — -bash — ⌘4

```
MacBook-Pro-2:lecture_examples michaelrohs$ make output
gcc -std=c99 -Wall -Werror -Wpointer-arith -Wfatal-errors
-g output.c -L../lib -lprog1 -iquote../lib -o output
MacBook-Pro-2:lecture_examples michaelrohs$ ./output
123456
3.14159
hello world
MacBook-Pro-2:lecture_examples michaelrohs$
```

output.c

Generate and Output Random Numbers
(Programming I C Library)

#include "base.h"

int	**i_rnd** (int i) Return a random int between in the interval [0,i). More...
double	**d_rnd** (double i) Return a random double between in the interval [0,i). More...

- Print a random integer number from the interval [0,100)

```
printiln(i_rnd(100));
```

- Print a random double-precision floating point number from the interval [0, 3.14)

```
printdln(d_rnd(3.14));
```

Input
(Programming I C Library)

- Input an integer number, terminated by a line break, store in variable i, multiply by 2, print

```
int i = i_input();
printiln(2 * i);
```

- Input a double-precision floating-point number, terminated by a line break, store in a variable d, multiply by 1.5, print

```
double d = d_input();
printdln(1.5 * d);
```

- Shorter:

```
printiln(2 * i_input());
printdln(1.5 * d_input());
```

Input: Complete Program
(Programming I C Library)

```c
// Compile: make input
// Run: ./input
#include "base.h" // we use printiln, printdln, i_input, d_input

int main(void) {
   int i = i_input();
   printiln(2 * i);
   double d = d_input();
   printdln(1.5 * d);
   return 0;
}
```

shorter

```c
int main(void) {
   printiln(2 * i_input());
   printdln(1.5 * d_input());
   return 0;
}
```

script_examples — -bash — ⌘1

```
MacBook-Pro-2:script_examples michaelrohs$ make input
gcc -std=c99 -Wall -Werror -Wpointer-arith -Wfatal-errors -g input.c -L../lib -lprog1 -iquote../lib -o input
MacBook-Pro-2:script_examples michaelrohs$ ./input
123
246
1.5
2.25
MacBook-Pro-2:script_examples michaelrohs$
```

input.c

Input a String
(Programming I C Library)

Read a single line of input from the console

```
String s = s_input(100); // reads at most 100 characters
```

- Print the input followed by a line break

```
prints("Your input was: ");
println(s);
```

- Print the number of characters

```
prints("Number of characters: ");
printiln(s_length(s));
```

#include "base.h"

String s_input (int n)
Read at most n-1 characters into a newly allocated string. More...

int **s_length** (**String** s)
Return the length of the string (number of characters).

Input and Output: Complete Program
(Programming I C Library)

```c
// Compile: make input_output
// Run: ./input_output
#include "base.h"   // we use prints, printsln, printiln, s_input
#include "string.h" // we use the String type and s_length

int main(void) {
   String s = s_input(100); // reads at most 100 characters

   prints("Your input was: ");
   printsln(s);

   prints("Number of characters: ");
   printiln(s_length(s));

   return 0;
}
```

input_output.c

```
script_examples — -bash — ⌘2
MacBook-Pro-2:script_examples michaelrohs$ make line_in_out
gcc -std=c99 -Wall -Werror -Wpointer-arith -Wfatal-errors -g
 line_in_out.c -L../lib -lprog1 -iquote../lib -o line_in_out
MacBook-Pro-2:script_examples michaelrohs$ ./line_in_out
my input
Your input was: my input
Number of characters: 8
MacBook-Pro-2:script_examples michaelrohs$
```

test_equal_* Functions
(Programming I C Library)

- Comparing actual to expected function results
  - Different check functions for different types
  - Checks whether actual result is identical/close to the expected one
  - Prints success/failure message, counts successes/failures
- Integers: test_equal_i(actual, expected);
  - `test_equal_i(s_length("hello"), 5);`
- Doubles: test_within_d(actual, expected, tolerance);
  - `test_within_d(sqrt(2.0), 1.414, 0.01);`
- Strings: test_equal_s(actual, expected);
  - `test_equal_s(s_concat("hello", "world"), "helloworld");`
- Booleans: test_equal_b(actual, expected);
  - `test_equal_b(s_contains("world", "rl"), true);`

test_equal_* Functions: Examples as Program
(Programming I C Library)

```c
// Compile: make test_equal
// Run: ./test_equal

#include "base.h"   // check_..., sqrt
#include "string.h" // s_length, s_concat, s_contains

int main(void) {
  test_equal_i(s_length("hello"), 5);
  test_within_d(sqrt(2.0), 1.414, 0.01);
  test_equal_s(s_concat("hello", "world"), "helloworld");
  test_equal_b(s_contains("world", "rl"), true);
  return 0;
}
```

test_equal.c

Recipe for Atomic Data: Problem Statement

- Write down the problem statement as a comment.
  - What is the relevant information?
  - What should the function do with the data?

- Example

```
/*
Design a function that computes weekly wages with overtime from
hours worked. The hourly rate is 10 €/hour. Regular working time is
40 hours/week. Overtime is paid 150% of the normal rate of pay.
*/
```

Examples with Expected Results (Test Function)

- Comparison of actual and expected result
- Examples (wage is 10 € per hour, 15 € for overtime)

```
void hours_to_wages_test() {
  test_equal_i(hours_to_wages(0), 0);
  test_equal_i(hours_to_wages(20), 20 * 1000);
  test_equal_i(hours_to_wages(39), 39 * 1000);
  test_equal_i(hours_to_wages(40), 40 * 1000);
  test_equal_i(hours_to_wages(41), 40 * 1000 + 1 * 1500);
  test_equal_i(hours_to_wages(45), 40 * 1000 + 5 * 1500);
}
```

Function Body

- Implementation of the function
- Example

```
// Computes the wage in cents given the number of hours worked.
int hours_to_wages(int hours) { // returns cents
    if (hours <= 40) {
        return hours * 1000;
    } else {
        return 40 * 1000 + (hours - 40) * 1500;
    }
}
```

Testing

- Main function call test function

```
int main(void) {
    hours_to_wages_test();
    return 0;
}
```

- Test results

```
wages.c, line 20: check passed
wages.c, line 21: check passed
wages.c, line 22: check passed
wages.c, line 23: check passed
wages.c, line 24: check passed
wages.c, line 25: check passed
All 6 tests passed!
```

LOOPS

While-Loop

- While-loop

```
while (condition) {
   statements
}
```

execute statements as long as condition is true

- Semantics
  - Check condition
  - If condition is false continue after loop
  - Otherwise execute **statements**
    and then repeat (check condition again, …)
- Implication
  - **statements** are <u>never</u> executed if condition is initially false

While-Loop and For-Loop

- While-loop

```
expr₁;
while (expr₂) {
   statements
   expr₃;
}
```

equivalent to

- For loop

initialize check update

```
for (expr₁; expr₂; expr₃) {
   statements
}
```

- Typical pattern

```
for (i=0; i<10; i++) {
    ...
}
```

- Semantics
  - Check expression $expr_2$
  - If $expr_2$ is false (0) go on after loop
  - Otherwise execute statements and then repeat (check $expr_2$ again, ...)

Example: Fahrenheit-Celsius Table (While-Loop)

```c
#include "base.h"
int main(void) {
   double lower = 0.0;
   double upper = 300.0;
   double step = 20.0;
   double f = lower;
   while (f <= upper) {
      double c = (f - 32.0) * 5.0 / 9.0;
      printd(f);
      prints("   ");
      printd(c);
      println();
      f += step;
   }
   return 0;
}
```

fctable.c

```
0    -17.7778
20    -6.66667
40    4.44444
60    15.5556
80    26.6667
100    37.7778
120    48.8889
140    60
160    71.1111
180    82.2222
200    93.3333
220    104.444
240    115.556
260    126.667
280    137.778
300    148.889
```

Example: Fahrenheit-Celsius Table (For-Loop)

```c
#include "base.h"
int main(void) {
   double lower = 0.0;
   double upper = 300.0;
   double step = 20.0;
   double f;
   for (f = lower; f <= upper; f += step) {
      double c = (f - 32.0) * 5.0 / 9.0;
      printd(f);
      prints("   ");
      printd(c);
      println();
   }
   return 0;
}
```

fctable.c

```
0    -17.7778
20    -6.66667
40    4.44444
60    15.5556
80    26.6667
100    37.7778
120    48.8889
140    60
160    71.1111
180    82.2222
200    93.3333
220    104.444
240    115.556
260    126.667
280    137.778
300    148.889
```

Example: Nested For-Loops

- Convenient syntax for nested loops
- Example: Print all pairs of numbers (i,j) with i < j

```
int n = 5;
for (int i = 1; i <= n; i++) {
   for (int j = i + 1; j <= n; j++) {
      prints("(");
      printi(i);
      prints(", ");
      printi(j);
      printsln(")");
   }
}
```

nested_loops.c

```
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(2, 3)
(2, 4)
(2, 5)
(3, 4)
(3, 5)
(4, 5)
```

Do-While-Loop

- Do-while-loop

```
do {
   statements
} while (expression);
```

do-while executes loop body at least once

- Semantics
  1. Execute statements
  2. Then check expression
  3. If expression is 0 (false) exit loop
  4. Otherwise continue with step 1

Break

- Break leaves the innermost loop or switch statement
  - Use sparingly, can make programs hard to read
- Break example: execute commands and finish

```
while (true) {
   prints("> ");
   String s = s_input(100); // read up to 100 characters
   if (s_contains(s, "exit")) break; // exit loop
   prints("executing ");
   println(s);
}
println("finished");
```

```
> start
executing start
> accelerate
executing accelerate
> jump
executing jump
> exit
finished
```

breaker.c

Without Break

- Same example, but without break

```
prints("> ");
String s = s_input(100);
while (!s_contains(s, "exit")) {
   prints("executing ");
   println(s);
   prints("> ");
   s = s_input(100);
}
println("finished");
```

```
> start
executing start
> accelerate
executing accelerate
> jump
executing jump
> exit
finished
```

breaker.c

Continue

- Skip the rest of the loop body, do not leave the loop
  - Use sparingly, can make programs hard to read
- Only process even values

```
for (i = 1; i <= 50; i++) {
    // skip odd values (least significant bit set)
    if ((i & 1) == 1) continue; // or (i % 2) == 1
    // assert: i is even
    // process even values...
    ...
}
```

Without Continue

- Only process even values

```
for (i = 1; i <= 50; i++) {
    // only use even values (least significant bit clear)
    if ((i & 1) == 0) {
        // assert: i is even
        // process even values...
        ...
    }
}
```

Better way to enumerate even integers in {1, ..., 50}?

Enumeration of Even Numbers in {1, ..., 50}

Only process even values

```
for (i = 2; i <= 50; i += 2) {
        ...
}
```

C Keywords (ANSI C / ISO C89)

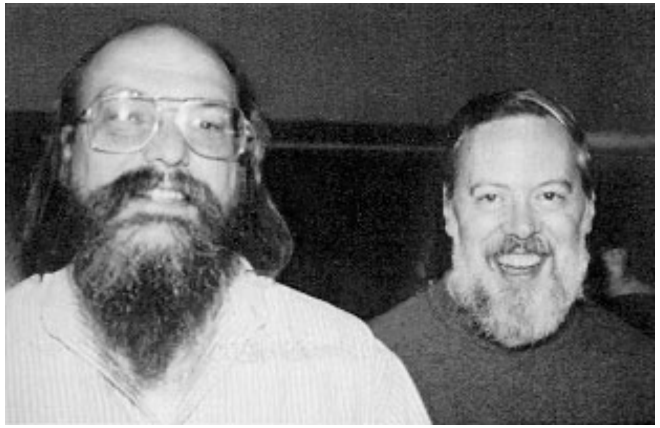
auto	**double**	**int**	struct
break	**else**	long	switch
case	enum	register	typedef
char	extern	**return**	union
const	float	short	unsigned
continue	for	signed	**void**
default	goto	sizeof	volatile
do	if	static	while

Summary

- Execution model of C
  - C closer to hardware, typical CPUs: register machines
- Variables and constants: declaration, definition
  - Variables need to be declared (type) and defined (value)
- Conditional execution: if statement
  - if … else, dangling else
- Functions: declaration, definition
  - Function header declares signature, function body
- Programming I C library
  - Simplify initial C programming
- Atomic Data (in C)
- Loops: while, for, do-while

Leisure Reading

- TIOBE index of the popularity of programming languages
  - https://www.tiobe.com/tiobe-index/
- Opinions on why C is still used
  - http://programmers.stackexchange.com/questions/103897/is-the-c-programming-language-still-used
- Dennis M. Ritchie: The Development of the C Language
  - http://csapp.cs.cmu.edu/2e/docs/chistory.html
  - creator of C
  - article difficult to understand with limited C knowledge
  - maybe read at the end of the semester



Ken Thompson (left) and Dennis Ritchie (right)

Turing Award 1983