Übersicht über das Modul PBI

Inhalt

1. grundlegenden Konzepte und Methoden der imperativen und objektorientierten Programmierung
2. Anwendungsbeispiele aus der Bioinformatik

Lernziele: Sie erwerben ...

1. Anwenderkenntnisse in Linux ($\Rightarrow$ Übungen)
2. die Fähigkeit zum Umgang mit grundlegenden Entwicklungswerkzeugen wie Editoren, Compilern und Debuggern
3. grundlegende Kenntnisse der Programmiersprachen Ruby und C
4. die Fähigkeit Softwarelösungen für kleinere und mittlere Probleme der Bioinformatik zu entwickeln
5. Kenntnisse des Klassenkonzepts in Ruby
6. Anwenderkenntnisse der dynamische Speicherverwaltung in C für Sequenzen, Bäume, Graphen

Übersicht über das Modul PBI

Teil 1: Programmierkurs C (ca. 7 Wochen)

Teil 2: Programmierkurs in Ruby (ca. 7 Wochen)

C-Programming Course
Programming in Bioinformatics
Lecture notes on a course held in the winter 2013/2014

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

October 13, 2013

Contents: sections

Contents: sections (cont.)

Contents: sections and subsections

Contents: sections and subsections (cont.)

Contents: sections and subsections (cont.)

Contents: sections and subsections (cont.)

Contents: sections and subsections (cont.)

Contents: sections and subsections (cont.)

Contents: sections and subsections (cont.)

Contents: sections and subsections (cont.)

- Bitfields

9 Important software libraries
- Dynamic Memory Management
- File operations
- Standard files
- Flushing file buffers and positioning the file
- Typed files
- Counting characters for input read from a file
- Mathematical Functions in C
- Recursive data structures
- Higher order functions
- Threads for concurrent programming

The first C-program

- Goal: Output the text "hello world"

```
/* hello world program */

#include <stdio.h>          /* declations for input/output */

int main()                  /* main: first function called */
{
  printf("hello world\n"); /* call function for output */
  return 0;                 /* end main */
}
```

“Hello World” in pieces

- The symbols `/*` and `*/` enclose a comment
- are ignored by the compiler
- comments improve the readability of the program

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

- the file `stdio.h` is included
- this contains definitions for input and output
- lines which start with #, are statements for the preprocessor i.e. they are evaluated before the compilation

“Hello World” in pieces (cont.)

```
int main ()
```

- Definition of a function with name `main`, a return value of type `int` and no parameters (due to `()`)
- a function contains a sequence of statements
- the function with name `main` is the first executed
- `{ ... }` encloses the sequence of statements for the `main`-functions
- in general: they build a block of statements

“Hello World” in pieces (cont.)

```
printf("hello world\n");
```

- call of the function with name `printf` and parameter “hello world”
- The function `printf` is declared in the file `stdio.h` and shows the given parameter on standard output channel
- `\n`: special character for new line
- declarations, statements etc must be closed with a semicolon `;`

“Hello World” in pieces (cont.)

```
return 0;
```

- ends the function and returns the value to the calling environment:
  - calling environment for `main` is the operating system which runs the program
  - calling environment for all other functions is the function in which they are called
- return values are the result of the function
- return value 0 in `main` means regular end of program

The different steps of compilation

This section closely follows the article `http://www.thegeekstuff.com/2011/10/c-program-to-an-executable/`

Preprocessing

- Macro substitution
- Comments are stripped off
- Expansion of the included files

`gcc -Wall -E hello-world.c`

outputs the result of the preprocessing-step

`gcc -Wall -save-temps hello-world.c -o hello-world`

creates three files of which the first two are normally not created

- `hello-world.i` contains the result of the preprocessing-step
- `hello-world.s` see below
- `hello-world.o` see below

The different steps of compilation (cont.)

Compilation

- In this phase, the input `hello-world.i` undergoes syntax and semantics checks
- Creates the file `hello-world.s` which contains low level instructions representing the program:
- These instructions are still independent of the concrete processor

The different steps of compilation (cont.)

Assembly

- Translates `hello-world.s` into machine code (in ELF format) stored in `hello-world.o` (the object file).
- This is what the CPU can understand, but not yet execute.
- the used format is ELF which stands for *executable and linkable format*.

The different steps of compilation (cont.)

Linking

- Final stage at which all the linking of function calls with their definitions are done.
- Until this stage the compiler does not know about the definition of functions like `printf()`.
- Until the compiler knows exactly where all of these functions are implemented, it simply uses a place-holder for the function call.
- It is at this stage, that the definition of `printf()` is resolved and the actual address of the function printf() is plugged in.
- The linker also does some extra work: it combines some extra code to our program that is required when the program starts and when the program ends.

The GNU C-Compiler gcc

- combines all phases of the compiler from the preprocessor to the linker
- Call: `gcc` [*option* | *filename*]
- important options
  1. `-o` *filename*: name of output file
  2. `-c`: only compile, no linking
  3. `-Wall`: show all warnings (errors are shown anyway); should always be used, as the warnings provide useful hints to strange code lines which may be a source for an unwanted behavior
- complete documentation of `gcc`:

  ```
  man gcc
  info gcc
  http://gcc.gnu.org
  ```


The GNU C-Compiler gcc (cont.)

- example 1:

  ```
  gcc -Wall program.c -o program.x
  ```

- creates executable program `program.x` (Compiler and Linker are called in a row)
- example 2:

  ```
  gcc -Wall -c module1.c -o module1.o
  gcc -Wall -c module2.c -o module2.o
  ```

- creates the object files `module1.o` and `module2.o`.
- By calling the linker in the next step, we obtain an executable program `myprogram.x`:

  ```
  gcc module1.o module2.o -o myprogram.x
  ```

Properties and application areas for C

- high-level programming language with some features close to the machine (e.g. memory management)
- many operating systems are written in C (e.g. Linux Kernel)
- efficient programs are possible
- full control over the resources (e.g. memory) but also full risk in doing something wrong (e.g. invalid memory access)
- many mature C-software libraries in diverse areas are available (e.g. GNU Scientific library)
- more modern programming concepts available in other languages, which e.g. better support object-oriented programming

Symbols in the programming language C

- alphanumeric symbols: a, A, b, B, ..., 1, 2, ...
- white spaces and separators: e.g. space, tabulator, new line
- Special symbols and control characters:

`\n`	new line
`\t`	tabulator
`\'`	'
`\"`	"
\ooo	octal number (with o being a digit between 0 and 7)
\xhh	hexadecimal number (with `h` being a digit or one of the letters `a`, `b`, `c`, `d`, `e`, `f`)

Operators and keywords

38 operators

```
[ ]   ( )   { }   .   ->   ?   :   ,   sizeof
++    --    &     *   +    -   /   %   ~
!     <<    >>    <   >    <=  =>  ==  !=
^     &&    *=    /=  +=   =   -=  <<= >>= &=  ^=
```

32 keywords

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

Identifier

- identifiers are used for referring to a number of things like variables, functions, types
- an identifier consist of letters, digits, and the symbol _, with the first symbol not being a digit
- keywords (see above) can not be used as identifiers
- identifier can be arbitrary long, but only the first 31 characters of an identifier are significant

⇒ so, a C-compiler may treat

`this_is_an_identifier_for_something`

and

`this_is_an_identifier_for_something_else`

as the same.

Variables and datatypes

- Variable (in computer science): memory cell to store values
- C is a strongly typed language, i.e. each variable must be of a certain datatype determined at compile time
- variables must be declared (i.e. it must be given a name and a type) before their first use

$\Rightarrow$ so the size of each variable (in bytes) can be determined at compile time

example:

```
int number;          /* declare number as integer variable */
number = 12;         /* use of number to store 12 */
```


Elementary datatypes and their variations

Elementary Datatypes

`char`	one byte or a character
`int`	integer
`float`	small floating point number
`double`	large floating point number

Variations

`short int`	small integer using less space
`long int`	large integer using (usually more space)
`signed`	with sign (for `int` and `char`)
`unsigned`	without sign (for `int` and `char`)
`long double`	very large floating point number

Ranges of values

- ranges of values are implementation and machine dependent
- the ranges are defined in `limits.h` and `floats.h`
- For example: `gcc` on `x86_64` (64 bits)

Type	Min	Max	size (byte)
char	-128	127	1
short int	-2^{15}	2^{15}	2
int	-2^{31}	2^{31}	4
long int	-2^{63}	$2^{63}-1$	8
unsigned int	0	$2^{32}-1$	4
unsigned long	0	$2^{64}-1$	8

Type T	Min	Max	Decimal-digits	precision ϵ_T	size byte
float	$1.17549 * 10^{-38}$	$3.40282 * 10^{+38}$	6	$1.19209 * 10^{-07}$	4
double	$2.22507 * 10^{-308}$	$1.79769 * 10^{+308}$	15	$2.222045 * 10^{-16}$	8
long double	$3.3621 * 10^{-4932}$	$1.18973 * 10^{+4932}$	18	$1.0842 * 10^{-19}$	12

$\epsilon_T = \max\{f \text{ of type } T \mid 1.0_T + f = 1.0_T\}$, 1.0_T means 1 of type T

Integer constants

- Decimal
  - `1234` is of type `int`
  - `1234l` and `1234L` are of type `long int`
  - `123456789` is of type `long int`
- Octal (beginning with `0`)
  - `037` stands for decimal 31
- hexadecimal (starts with `0x` or `0X`)
  - `0x1f` and `0X1F` stands for decimal 31
  - `0XFUL` is of type `unsigned long` and stands for decimal 15

Floating point constants

with decimal point

- `1.234` is of type `double`
- `1.234f` and `1.234F` are of type `float`
- `1.234l` is of type `long double`

with exponent

- `1234e-3` is of type `double` and means $1234 \cdot 10^{-3}$, i.e. 1.234
- `1.234e2f` is of type `float` and means 123.4

Character constants

- in single quotes, e.g. `'a'`
- stands for numerical value of the character in the given character set
- note that `'7'` $=$ `55` $\neq$ `7`
- special symbols:

\a	bell	\\	\(backslash)
\b	backspace	\?	? (question mark)
\f	form feed	\'	' (quote)
\n	new line	\"	" (double quote)
\r	carriage return	\ooo	character with octal number
\t	tabulator	\xhhh	character with hexadecimal value
\v	vertical tab		

- Strings in double quotes, e.g. `"Hello World\n"`

Example: Variables and constants

```
short int int2;
int int4;
double pd;
char c1, c2;

int2 = 12;
int4 = 45;
pd = 4.5e-3;
c1 = 'a';
c2 = 97;
```


Memory

- RAM stands for Random Access Memory
  - reading and writing possible
  - Memory cells are accessed by sequentially numbered addresses
  - Each memory cell can be addressed
- A memory cell stores one byte
  - 1 byte consists of 8 bits
  - each bit can take the value 0 or 1
  - k bits can store 2^k different values in the range from 0 to $2^k - 1$

  $\Rightarrow$ 1 byte can store $2^8 = 256$ values in the range from 0 to 255
  - 1 Kbyte $= 2^{10} = 1024$ byte
  - 1 Mbyte $= 2^{20} = 1024$ Kbyte
  - 1 Gbyte $= 2^{30} = 1024$ Mbyte
  - 1 Tbyte $= 2^{40} = 1024$ Gbyte

Memory (cont.)

– the following scheme shows, how the variables from the declarations

```
short int int2; int int4; double dp; char c1, c2;
```

– are stored beginning with the address 4713

4711	4712	4713 int2	4714 int2	4715 int4	4716 int4	4717 int4	4718 int4
4719 dp	4720 dp	4721 dp	4722 dp	4723 dp	4724 dp	4725 dp	4726 dp
4727 c1	4728 c2	4729	4730	4731	4732	4733	4734

– note: the type of variables cannot be determined from the contents of the memory cells

Variables, memory and addresses

Addresses

– The address of a variable is the number of the first memory cell, at which the value of the variable is stored.

– Example: the variable `int2` is stored in two bytes at address 4713 and 4714. So the address of `int2` is 4713.

– The number of memory cells following the address at which the value is stored is derived from the type at compile time

– the address-operator `&` delivers the address of a variable

– example: `&int4` delivers the address of `int4`

Variables, memory and addresses (cont.)

Pointer

- A pointer is a variable, which stores the address of a variable
- pointer refer to a variable of a specific type
- declaration of a pointer: `Type *pointervariablename;`
- example: `double *dp;`
- note: declaration of pointer only reserves space for storing an address, but not for the value it refers to

Variables, memory and addresses (cont.)

- The dereferencing operator $*$ allows to access the value of the memory cells, to which a pointer refers
- In the following examples we assume that a pointer requires 4 bytes

```
short int n, m, *ptr;

m = 15;       /* store 15 in m */
ptr = &n;     /* store address of n */
*ptr = 42;    /* contents of var. referred to by ptr becomes 42 */
if (m < n)
  return EXIT_SUCCESS;
return EXIT_FAILURE;
```

4711	4712 42	4713 42	4714 15	4715 15
4716 4712	4717 4712	4718 4712	4719 4712	4720

Application of pointers

pointers are used

- when programming close to the hardware: if one has to directly access a fixed memory cell
- for dynamic allocation of memory:
  - if, at compile time, it is not determined how much space is required to store certain structured values (like e.g. lists or trees)
  - there are special functions by which the memory is allocated.
  - the first address of the memory cell is delivered by the allocation function
  - a complementary function frees the space given this address

Constants

If a value does not change during the lifetime of a program it can be defined as a constant:

- keyword `const` before variable declaration
  - creates variable, which cannot be modified

```
const double PI = 3.14159265;
...
printf("Pi = %g\n",PI);
```

- macro-substitution using `#define`
  - replace the name defined in macro by the given value
  - replacement is done in the preprocessing stage

```
#define PI 3.14159265
...
printf("Pi = %g\n",PI);
```

Some important operators

Expressions are build from constants, variables and operators. Lets consider the most important.

- variable assignment: `=`

```
int n;
n = 15;
```

- basic arithmetic operations: `+, -, *, /`

```
int n, m;
n = 15;
m = n+3;
```

- Note: multiplication/division binds more tightly than addition/subtraction (Punkt vor Strich-Rechnung)
- use brackets to clarify in which order the evaluation is done

```
short n; int m;
n = 15;
m = 5 * (n+3) - 4;
```


Implicit conversion of types

- what if there are two operands of different types, to which an arithmetic operator is applied:
  e.g. `short n; int m;`
  what is the type of the expression `n + m`

⇒ implicit type conversion (to the larger type) applies

- another example:

```
int i;
float f;
i = 15;

f = 1.23;
f = i + f;
```

the result of the addition is of type `float`

Rules for implicit type conversion

type op1	type op2	type result	kind of arithmetic
char	int	int	integer arithmetic
int	long	long	integer arithmetic
char/int/long	float/double	double	floating point arithmetic

- pointers can be turned into integers
- in assignments of variables of a type with a larger range of value to a variable with a smaller range of values, it is tried to preserve the value

```
char i;
double f;
f = 1.23;
i = f;          /* i becomes 1 */
f = 2.3e12;
i = f;          /* not defined */
```


Importance of functions

- programs often contain parts executed more than once
- these parts can be defined in a function, which is called at the corresponding places
- advantages:
  - shorter, less redundant programs
  - easier maintenance (debugging, extension)

Calling functions

- functions can have parameters which have to be supplied when calling the function

```
printf("hello world\n");
```

- functions can have return value, which may be ignored

```
double x;
x = sin(2);     /* store return value in x */
(void) sin(3); /* ignore return value */
```


Functions for char

- declarations of functions in `ctype.h`
- test functions

```
int isalpha(int c)  /* checks for an alphabetic character */
int isdigit(int c)  /* checks for a digit (0 through 9) */
int islower(int c)  /* checks for a lower-case character */
int isprint(int c)  /* checks for any printable character */
int isspace(int c)  /* checks for white-space characters */
int isupper(int c)  /* checks for an uppercase character */
```

- conversion functions

```
int toupper(int c)  /* converts  the letter c to upper case */
int tolower(int c)  /* converts the letter c to lower case */
```

Functions for input/output

- file `/usr/include/stdio.h` declares input and output functions
- use `#include <stdio.h>`
- operations for file input and output (including STDIN, STDOUT, STDERR)
- part of standard libraries and software development environment
- manual pages are available

putchar and getchar

- `putchar`: write character to STDOUT
- `getchar`: read character from STDIN

```
#include <stdio.h>

int main()
{
  char z;
  z = getchar();  /* read character */
  putchar(z);     /* write character to stdout */
  return 0;
}
```

Output with printf

- formatted output of variables and constants of basic types
- returns number of actually output values
- syntax: `int printf(formatstring[,parameter])`
- example:

```
#include <stdio.h>
...
int age;
age = 15;
printf("Ted is %d years old\n",age);
```


The format string of printf

- The format string consists of output text and/or format specifications
- text is output verbatim
- format consists of %-symbol and character denoting the basic data type to output
- format specification is replaced by the next not yet processed parameter of `printf`

The most important format specifications

code	type of value	show
`%c`	`int` ≤ 255	character
`%d`	`int`	decimal integer
`%d`	`char`	decimal integer
`%hd`	`short int`	decimal integer
`%i`	same as `%d`	
`%u`	`unsigned int`	decimal int without sign
`%s`	`char *`	sequence of characters up to first `\0`.
`%f`	`double`	floating point notation
`%e, %E`	`double`	exponential notation with lower/upper case `e`
`%%`	none	`%`

Extended format specifications

- allows column by column output of numerical values
- allows to restrict the number of decimal places
- syntax: %[*flags*] [*width*] [.*precision*] [*.l*] *type*
- *flags:*
  - -: left-adjusted output
  - +: signed output even for positive numbers
  - 0: padding with 0 for given width
  - *blank*: empty space before positive number
- *width*: minimum width of output field
  - if output is larger: width is ignored
  - if output is smaller: left-adjusted blanks before output
- .*precision*: for floating point number gives number of decimal places
- `l` means to use the `long`-version of the given data type
- `type` is a simple format character (`d,c,i,s,f`)

A printf-example

```
char c = 'S';
int i = 3242, j = -12345;
const char *string = "Programming";
double d = 1234.56789;

printf("c=%c as left adjusted number \"%-5d\"\n",c,(int) c);
printf("i and j decimal \"%+8d\" \"%-8d\"\n",i,j);
printf("i decimal with leading zeros \"%08d\"\n",i);
printf("string=\"%s\"\n",string);
printf("d with sign and 5 decimal places: %+.5f\n",d);
printf("d with exponent and 5 decimal places: %.5e\n",d);
```

delivers the following output:

```
c=S as left adjusted number "83   "
i and j decimal "   +3242" "-12345  "
i decimal with leading 0s "00003242"
string="Programming"
d with sign and 5 decimal places: +1234.56789
d with exponent and 5 decimal places: 1.23457e+03
```


Input with scanf

- is used for formatted input of variables
- return value: number of successfully read input values (allows for detecting corrupted input)
- syntax: `int scanf(formatstring[,parameter])`
- the format string consists of format specifiers, white spaces and other symbols
  - format specifiers consist of a `%` followed by a symbol denoting the data type expected in the input (as for `printf`)
  - a single white space represents a white space of any size
  - other symbols only match themselves
  - inputs are stored at the addresses given in the parameters (thus parameters are pointers)

A scanf-example

```
int age, x, y;
printf("How old is Ted? ");
if (scanf("%d",&age) == 1)
  printf("Ted is %d years old\n",age);
printf("Addition? ");
if (scanf("%d plus %d is",&x,&y) == 2)
  printf("%d\n",x+y);
```

Note: the parameters `&age`, `&x` and `&y` are address to variables output for the given input:

```
How old is Ted? 12
Ted is 12 years old
Addition? 3 plus 5 is
8
```


Internal representation of values

bits

- smallest unit of information
- binary: can take value 0 or 1 (which can be interpreted as false and true)
- all information stored in the computer is represented by bits which are combined into bytes
- with n bits one can represent 2^n values, e.g. the values from 0 to $2^n - 1$ or from -2^{n-1} to $2^{n-1} - 1$.

bytes

- 1 byte = 8 bits
- 1 byte can store $2^8 = 256$ different values
- basic data type `unsigned char` has 1 byte and can store the numbers from 0 to 255 (which may be interpreted as characters).

Representation of numbers

meaning of 4711 as decimal number

$4711 = 4 \cdot 10^3 + 7 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0$

general representation to base $B \geq 2$:

$b_{n-1} b_{n-2} \ldots b_1 b_0$ stands for
$b_{n-1} \cdot B^{n-1} + b_{n-2} \cdot B^{n-2} + \ldots + b_1 \cdot B^1 + b_0 \cdot B^0$

important numbering systems for computer science:

- binary system (base 2, use digits $0, 1$)
- octal system (base 8, use digits $0, \ldots, 7$)
- hexadecimal system (base 16, use digits $0, \ldots, 9, a, b, c, d, e, f$)

Operators, operands, expressions

operators

- are used for combining operands
- example: +, *
- in C there are unary, binary and ternary operators

operands

- arguments of operators
- in C: constants, variables, function calls and expressions

expressions

example: `-c1 + c2 - 5 * (f(j)- 10)`

Expressions

inductive definition of expressions

- constants
- variables
- function calls
- A expression and ϕ unary prefix-operator $\Rightarrow \phi A$ is expression
- A expression and ϕ unary postfix-operator $\Rightarrow A\phi$ is expression
- A, B expressions and ϕ binary infix-operator $\Rightarrow A\phi B$ is expression
- A, B, C expressions and ϕ/ψ denote a ternary infix-operator $\Rightarrow A\phi B\psi C$ is expression
- A expression $\Rightarrow (A)$ is expression

return value

each expression has a return value, namely the value determined by evaluating the expression

Order of evaluation of expressions

grouping of expressions

- expression in brackets
- unary postfix operators
- unary prefix operators
- binary and ternary operator according to the precedence rules given in the following tables:

operator precedence

- operators are grouped
- the higher the rank of the group the operator occurs in, the more tightly it binds
- operators of the same precedence have precedence according to the associativity defined (either from left to right ($l \Rightarrow r$) or from right to left ($r \Rightarrow l$))

Groups of operators I

operators	meaning	associativity

`( )`
`[ ]`
`->`
`.`
	function/expression bracket
	array indexing
	pointer to structure element
	structure element
		$l \Rightarrow r$

`!`
`~`
`++`
`--`
`-`
`(type)`
`*`
`&`
`sizeof`
	negation
	bit complement
	increment
	decrement
	negative sign
	type cast
	dereferencing pointer
	address of variable
	size of type or variable
		$r \Rightarrow l$

`*`
`/`
`%`
	multiplication
	division
	modulus
		$l \Rightarrow r$

`+` `-`	addition subtraction	$l \Rightarrow r$

Groups of operators II

operators	meaning	associativity
`<<` `>>`	shift left shift right	$l \Rightarrow r$

`<`
`<=`
`>`
`>=`
	lower than
	lower equal
	greater than
	greater equal
		$l \Rightarrow r$

`==` `!=`	equal not equal	$l \Rightarrow r$
`&`	bitwise and	$l \Rightarrow r$
`^`	bitwise exclusive or	$l \Rightarrow r$
`|`	bitwise inclusive or	$l \Rightarrow r$
`&&`	boolean and	$l \Rightarrow r$
`||`	boolean or	$l \Rightarrow r$
`?:`	conditional expression	$r \Rightarrow l$
`=` `op=`	assignments op $\in \{+, -, *, /, \%, \&, \hat{}, <<, >>\}$	$r \Rightarrow l$
`,`	comma operator	$l \Rightarrow r$

Operators by functional groups

arithmetic operators

`+, -, *, /, ...`

logical operators

`&&, ||`

assignment operators

`=, +=, -=, *=, /=`

bitwise operators

`&, |, <<, >>`

increment- and decrement-operators

`++, --`

other operators

`sizeof, ...`

relational operators

`==, =!, <, >, <=, >=`

Arithmetic operators

- binary (i.e.@ it has two arguments)
- no side effects, i.e. does not modify any variable
- left associative, e.g. `A + B + C` is interpreted as `(A + B)+ C`
- Let A and B be integer or floating point expressions.
  - `A + B`, `A - B`, `A * B`, `A / B` are arithmetic expressions
- when applying `/` to integers, the decimal place is cut (result is integer)
- modulus operator `A%B` only defined for integer operands (`char`, `int`, `long` and their variants)

Assignment operators

- right associative, i.e. `A = B = C` stands for `A = (B = C)`
- conversion, whenever operands are of different but compatible types
- is often combined with arithmetic operations
  - assignment after addition: `A += B` (stands for `A = A + B`)
  - assignment after subtraction: `A -= B` (stands for `A = A - B`)
  - Other possibilities: `A *= B`, `A /= B`, `A %= B`, `A &= B`, `A |= B`, `A <<= B`, `A >>= B`.

Increment and decrement operators

- increment operators increase the value of a variable
- decrement operators decrease the value of a variable

statement	return value	side effect
`x++`	`x`	`x` is incremented (`x = x+1`)
`++x`	`x+1`	`x` is incremented (`x = x+1`)
`x--`	`x`	`x` is decremented (`x = x-1`)
`--x`	`x-1`	`x` is decremented (`x = x-1`)

Example

```
int x = 5, y, z;
x++;   /* x stores 6 */
--x;   /* x stores 5 */
y = x++; /* y stores 5, x stores 6 */
z = --x; /* z stores 5, x stores 5 */
```

Relational operators

- relational operators compare two expressions
- boolean values in C:
  - no type `boolean`
  - but: integer value $\neq 0$ means `true` and 0 means `false`

Example:

`x = y < z`

- if `y` $<$ `z`, then `x` contains a value $\neq 0$
- if `y` $\geq$ `z`, then `x` contains the value 0

Operators:

==	test for equality	!=	test for inequality
<	test for lower than	<=	test for lower equal
>	test for greater than	>=	test for greater equal

Logical operators

- the logical operators !, `&&`, `||` are used for combining boolean values according to the following table

		negation	and	or
`x`	`y`	`!x`	`x && y`	`x || y`
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

false: value 0
true: value $\neq 0$

- relational and logical operators are important for control structures (like `if`-statements and `while`-loops)

Bitwise Operators

- bitwise operators work on the internal binary representation of the operands and combine them bit by bit:
- `&`: and, `|`: or, `^` exclusive-or, `~` inverting, `<<` left shift (fill with 0 bits), `>>`: right shift (fill with 0 bits)

x	y	~x	x & y	x | y	x ^ y
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Bitwise Operators (cont.)

Example

```
unsigned int x = 14, y = 19, z; /* x = binary 001110, y = binary
    010011 */
z = x & y;      /* z becomes binary 000010 (decimal 2) */
z = x | y;      /* z becomes binary 011111 (decimal 31) */
z = x << 1;     /* z becomes binary 011100 (decimal 28), mult 2 */
z = x >> 2;     /* z becomes binary 000011 (decimal 3), div 4 */
z = x & 3;      /* z becomes binary 000010 (decimal 2), mod 4 */
```

rules

- $a = b << c$ stands for $a = b * 2^c$
- $a = b >> c$ stands for $a = b/2^c$
- $a = b \ \& \ (2^c - 1)$ stands for $a = b \ \% \ 2^c$

Other operators

`sizeof`

returns internal size of a variable or type in size

- Example: `s = sizeof(float)`

comma-operator `A, B`

evaluates expressions `A` and `B` one after the other and delivers the result of `B`

conditional expression `A ? B : C`

first evaluate `A`; result is `true` ($\neq 0$) $\Rightarrow$ expression evaluates to `B`; result is `false` ($= 0$) $\Rightarrow$ expression evaluates to `C`

- Example (minimum of `x` and `y`): `m = x < y ? x : y;`

function call `A(B)`

call of the function `A` with parameter `B`

Other operators (cont.)

type cast `(type)A`

convert expression `A` into given `type`

address operator `&`

delivers address of variable

dereferencing operator `*`

delivers value of pointer or allows to store a value in a memory cell referenced by a pointer

array index operator `A[B]`

see arrays

structure element access operators `A.B` and `A->B`

see structures

Example for the dereferencing operators

```
int a = 0, b = 0, *intptr = &a, iter = 0;

while (*intptr < 100) /* dereferencing intptr */
{
  iter++;
  if (intptr == &a)   /* compare addresses */
  {
    intptr = &b;      /* intptr points to b */
  } else
  {
    intptr = &a;      /* intptr points to a */
  }
  (*intptr)++;        /* incrementing the contents of the memory
                         cell intptr points to */
}
printf("iterations=%d\n",iter);
```

– program performs 199 iterations

Blocks

– statements define the flow of the program
– expression becomes statement by adding a ;
– empty statement consists of ; only and is e.g. sometimes used in `while`-loops
– blocks: combination of declarations and statements between a pair `{` and `}` (e.g. in function declarations).
– a single block is considered a statement

Variables in blocks

- if declarations occurs in a block, they can only appear at the beginning of a block before the first statement
- declarations are only visible in the block (local variables) they appear in
- declarations shadow variables with the same name declared in the outer blocks

```
int i;
i = 5;
printf("before the block: %d\n",i);     /* output 5 */
{
  int i;
  i = 7;
  printf("in block: %d\n",i);         /* output 7 */
}
printf("after the block: %d\n",i);    /* output 5 */
```


Initialization of variables

- global variables (i.e. the variable is declared outside of all blocks) of standard types are initialized by 0
- local variables are not initialized, i.e. their value is not defined (they contain random values)
- give all variables a value before the variable is accessed for the first time:

```
{
  int i, j = 2, k = 5, l;
}
```

  - `i` and `l` remain uninitialized
  - `j` stores 2, `k` stores 5

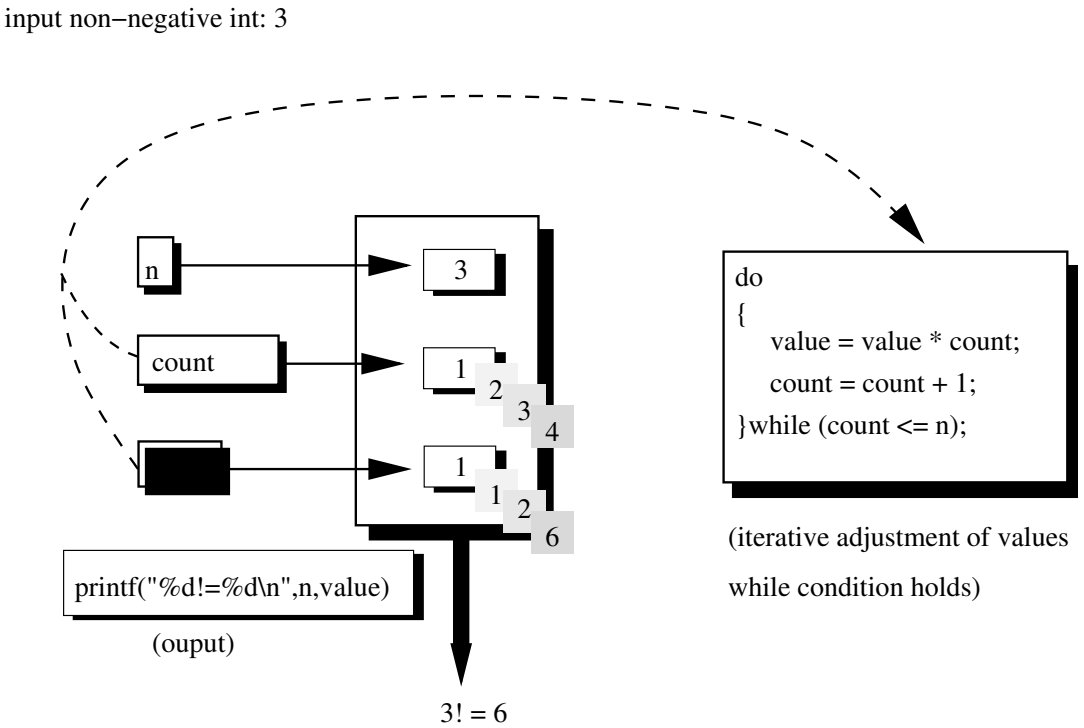
Example: Iterative computation of the faculty

```c
#include <stdio.h>                          /* use standard output */
#include <stdlib.h>                         /* use standard library */

int main(int argc, char *argv[])            /* first func. called */
{                                           /* start block */
  int value = 1, count = 1, n;              /* declare variables */

  printf("input non-negative int: ");       /* ask for input */
  if (scanf("%d",&n) != 1 || n < 0)         /* read non-neg. int */
  {                                         /* check if successful */
    fprintf(stderr,"incorrect input\n");    /* show error message */
    return EXIT_FAILURE;                    /* return with error */
  }
  do                                        /* iterate statements */
  {
    value = value * count;                  /* modify value */
    count = count + 1;                      /* increment count */
  } while (count <= n);                     /* continue while cond. */
  printf("%d!=%d\n",n,value);               /* print result */
  return EXIT_SUCCESS;                      /* return with success */
}                                           /* end block */
```


Iteration and Output



Global and local variables

global variables

- are visible everywhere

⇒ common use often lead to name clashes with other variables of the same name

- memory cells storing the value of the variable are used during the entire runtime of the program
- often lead to unstructured programs

local variables

- are visible in current block and in all blocks contained in the current block
- name clashes occur only rarely even for large programs
- recommended for structured programming
- are stored on the call stack and are freed as soon as the block was executed

Explicit type cast

- in some cases, the type of a variable does no satisfy the requirements of the context, e.g.
  - a variable of type `int` is part of an expression which needs to evaluate to a floating point value
  - a variable of type `int` should be interpreted as a character
- explicit type cast:
- `(type)var` means, that the variable `var` is interpreted as if it is of type `type`.

Example

```
int a = 5, b = 2;
printf("floating point division: %.2f\n", (double) a/b);
```

Control Structures

conditional statements

execute statements only if certain conditions are true:

- `if`, `else`, `switch`

loops

for iterating statements

- `while`, `do`, `for`

Jump instruction

continue execution with another statement somewhere in the program

- `break`, `continue`, `return`

The if-statement

- conditional statement: execute statement depending on the result of an expression

```
if (expression)
  block_of_statement1
else
  block_of_statement2
```

- if `expression` evaluates to true, then `block_of_statements1` is executed, otherwise `block_of_statements2`
- the `else`-part is optional

Example (check if number is even):

```
int i;
if (scanf("%d",&i) != 1)
{
  fprintf(stderr,"incorrect input\n");
  exit(EXIT_FAILURE);
}
if (i % 2 == 0)
  printf("even\n");
```

The if-statement (cont.)

- more than two alternatives are integrated by an `else if`-combination
- as soon as a condition is satisfied, the others are not tested and executed

```
int i, j;
if (scanf("%d %d",&i,&j) != 2) exit(EXIT_FAILURE);
if (i < j)
{
  printf("the first number is smaller.\n");
} else
{
  if (i > j)
  {
    printf("the first number is larger.\n");
  } else
  {
    printf("the numbers are equal.\n");
  }
}
```


Loops

- loops are used to iteratively execute a block of statements
  1. logical expressions define the termination condition
  2. in each iteration, the condition is checked. Once the result is false, the loop stops

The while-loop

- termination expression is evaluated before the statement is executed
- if it is true, then the first statement of the block is executed
- the block of statements is possibly never executed
- Syntax:

```
while (expression)
  block_of_statements
```

- Example (output of the numbers from 1 to 10):

```
int i = 1;
while (i <= 10)
  printf("%d ",i++);
```

- Example (wait for input character 'e')

```
int c;
while ((c = getchar()) != 'e') /* empty statement */ ;
```


The do-loop

- termination condition is tested after the execution of the statements of blocks
- if it is true, the statement of blocks is executed again
- the statement of block will be executed at least once
- Syntax:

```
do
  block_of_statements
while (expression)
```

- Example (output of all input characters, terminate after the character 'e' is input):

```
int c;
do
{
  c = getchar();
  putchar(c);
} while (c != 'e');
```

The for-loop

- mostly used if the number of executions is known in advance
- Syntax:

```
for (expression1; expression2; expression3)
  block_of_statements
```

  - `expression1`: initialization
  - `expression2`: test expressions
  - `expression3`: completion
  - any of the three expressions can be omitted
  - if `expression2` is omitted, then the termination condition is always true

The for-loop (cont.)

- Example (output of the numbers from 1 to 10):

```
int i;
for (i = 1; i<=10; i++)
{
  printf("%d ",i);
}
```

- Example (for-loop with two counters):

```
int i, j;
for (i = 1, j = 10; i<j; i++, j--)
{
  printf("(%d,%d) ",i,j);
}
```

Infinite Loops

- `while(1){ ...}` and `for (;;){ ...}` are loops, that never stop
- abandoning the loop is only possible with a jump-statement
- application:
  - termination condition must be tested in the middle of the loop
  - several or complicated termination conditions

Structured programming

Some useful rules

- think about the rational structure of the program
  - which parts can be put into functions?
  - which functions can be combined into modules?
- use meaningful names for variables and functions
- e.g. constants with meaningful names instead of incomprehensible numbers
- add comments to each functions
  - what is the output of the functions?
  - what do the parameters and return values mean?
  - what are the side effects?
  - ...

Structured programming (cont.)

– use indentation, to make complex control structures visible

```
int i = 0, j, s, c = 0;

while (c!='n')
{
  i++;
  s = 0;
  for (j = 1; j <= i; j++)
  {
    s += j;
  }
  printf("sum of 1 to %d: %d\ncontinue? ",i,s);
  c = getchar();
}
```


The switch-statement

– recall `if`-statement is used if only few alternatives or different conditions are to be checked

– `switch`-statement is used if many alternatives of the same expression lead to different statements

Syntax:

```
switch (expression)
{
  case constant1:
    block_of_statements1;
    break;
  case constant2:
    block_of_statements2;
    break;
  ...
  default:
    default_block_of_statements;
}
```

The switch-statement (cont.)

- in the `switch` it is checked, if the value of a given expression is equal to a certain constant which triggers the block of statements following the corresponding `case`.
- `break` stops the switch.
- if it is missing, then the statement following the next `case` is executed (fall through)
- if the value does not equal any of the given constants, then the statement following the optional `default`-tag is executed

The switch-statement (cont.)

```
int main()
{
  int inputchar, counta = 0, countc = 0, countg = 0, countt = 0;

  while ((inputchar = getchar()) != EOF)
  {
    switch (tolower(inputchar))
    {
      case 'a':
        counta++;
        break;
      case 'c':
        countc++;
        break;
      case 'g':
        countg++;
        break;
      case 't':
        countt++;
        break;
      case '\n':
        break;
```

The switch-statement (cont.)

```
      default:
        fprintf(stderr,"'%c' is not a base\n",inputchar);
        return 1;
    }
  }
  printf("a=%d, c=%d, g=%d, t=%d\n",counta,countc,countg,countt);
  return 0;
}
```


Jump-statements: break

- `break` can be used in all loops and in the `switch`-statement
- `break` causes a jump to the statement following the loop, or the `switch`-block
- leave an infinite loop:

```
while (1)
{
  int cc;
  cc = getchar();
  if (cc == 'e')
  {
    break;
  }
}
```

Jump-statements: continue and return

continue

- can be used in all loops
- it causes a jump to the next evaluation of the termination condition
- in the `for`-loop, it first causes an execution of the completion `expression3`, before the termination condition is checked

return

- `return` is used to immediately stop the execution of the function it occurs in and the statement following the function call is executed (see section on functions)

Motivation for using functions

- if a sequence of statements is used more than once, these should be combined into a function
- declaration of function does not execute the statement
- only the call of a function, referenced by the name of the function, triggers the execution of the sequence of statements in the function

Advantages:

- shorter programs
- easier maintenance (correction, extension, porting)

Disadvantage:

- none.

Declaration of functions

Declaration

- specify a function header followed by a block
- the program code is not executed, but just declared to the compiler

Syntax

- [*storage_class*] [*type*] *name* ([*parameterlist_and_definition*]) *block*

Example:

```
int mysquare(int x)
{
  return x * x;
}
```


Function call

Syntax:

- *functionname* ([*list_of_parameter*])

Execution of function

- jump to the place of the function definition
- execution of the statements in the function block
- continue the program with the first statement after the function call

Example:

```
int y;
y = mysquare(3);
```

Functions without parameters/return values

Declaration of function without parameters

- is very rare
- use empty list of parameters (`void`)

Declaration of functions without return values

- use keyword `void` instead of return type

Example:

```
void errormessage(void)
{
  printf("an error occurred\n");
}
```


Variables in functions

- in block of function declaration one can define local variables
- they are only visible in the block, i.e. in the function
- other parts of the program cannot directly access the local variables
- the memory for local variables
  - is allocated when the function is called
  - is freed when the function call is finished
- local variables are not automatically initialized

```
unsigned int sumup(unsigned int  i)
{
  unsigned int j, s = 0;

  for (j = 1; j <= i; j++)
  {
    s += j;
  }
  return s;
}
```

Function parameters

- parameters are values, which are transferred to the function by the calling program
- the function executes the statements of the function block with these values
- in the functions, parameters are handled as ordinary variables
- flexible concept: for each function call other values can be used
- function parameters and local variables must have different names
- parameters uniquely defined in function header by their name and type
- number and type of parameter in function call and function declaration must be consistent
- make a difference between
  - formal parameters which appear in declaration of function and
  - actual parameters which are used in the function call
- formal and actual parameter may, but do not necessarily have to have the same name

Parameter passing mechanisms

- In C, parameters are always passed "call by value", i.e.
  - in the function call, the memory cells are allocated according to the declaration of the formal parameters
  - the actual parameters are *copied* into these memory cells, i.e. the statements in the function body have no effect on the variables in the calling part of the program
  - the copies of the actual parameters become local variables

Parameter passing mechanisms (cont.)

- how can a function communicate values to the calling program?
  - global variables
  - return values
  - "call by reference"
- "call by reference" means that the address of the data to be modified is passed to the function

$\Rightarrow$ the data can directly be modified

- "call by reference" is emulated using C-pointers

Parameter passing mechanisms (cont.)

```
void inccbv(int i)   /* call by value */
{
  i++;
}
void inccbr(int *i)  /* call by reference via */
{                    /* call by value with int-ptr */
  (*i)++;
}
int main(void)
{
  int i = 0;
  inccbv(i);                          /* pass value i */
  printf("after inccbv: i=%d\n",i);   /* out: after inccbv: i=0 */
  inccbr(&i);                         /* pass address of i */
  printf("after inccbr: i=%d\n",i);   /* out: after inccbr: i=1 */
  return 0;
}
```

The return value

- with the return value, the computed value is passed to the calling program
- the type of the return value is given in the function definition
- basic datatypes and self-defined datatypes can be used as return types
- the return type `int` is the default
- functions with return values can be used as part of an expression which cannot be assigned a value

```
y = 3 * mysquare(x + 2) + 5;
```


The return statement

the `return` statement is used

- to immediately cancel the execution of the function
- to specify the value returned by the function to the calling part of the program

Syntax:

```
return expression;
```

Note:

the datatype of the return value must be compatible with the function declaration

Lifetime and visibility of functions

Lifetime

A function lives as long as the program it is part of.

Visibility

- function declaration is preceded by the keyword `static` ⇒ function is only visible in the same source file
- no keyword `static` ⇒ function is visible in all source files

Application

- in larger programs, which consist of more than one source files one declares all functions as `static` which are only used in a single file

⇒ prevents name conflicts and reduces work of loader

Recursion

Recursion

means self-reference

Example: definition of natural numbers

- 1 is a natural number
- if n is a natural number, then so is $n + 1$

recursive function

a function, which calls itself

Example: faculty

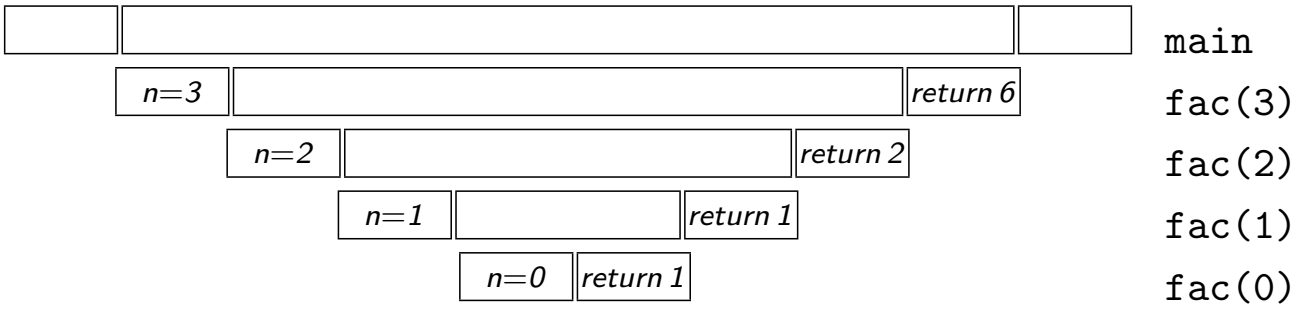
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

there must be a termination condition ($n = 0$ in this example)

Recursion (cont.)

```
static int fac(int n)
{
  if (n == 0)
    return 1;
  return n * fac(n-1);
}

int main(void)
{
  printf("3!=%d\n",fac(3));
  return 0;
}
```



Recursion vs. Iteration

Equivalent programming concepts

for each recursive solution there is an iterative solution (using loops)

recursive or iterative?

Criteria of decision:

- complexity of the different solutions
  - which solution is easier to program?
  - which solution is easier to understand?
- memory requirement
  - for each function call, all local variable require extra space

Self defined types

- introduce a new, alternative name for an existing type
- syntax: `typedef` *type Typename;*

Example:

```
typedef unsigned char Symbol; /* for small alphabets */
```

motivation:

- improve readability of programs
- improve portability of programs, since only the abstract datatypes must be adapted to the new environment

Example: alternative definition of the type above:

```
typedef unsigned short Symbol; /* for large alphabets */
```


Enumeration types

enumerations can be defined for variables, which can only take a few different values

Syntax

```
typedef enum
{
  Identifier1,
  Identifier2,
  ...
} Typename;
```

```
typedef enum
{
  Red,
  Yellow,
  Green
} Color;
```

Enumeration types (cont.)

internal representation:

- the values of an enumeration type are stored as values of type `int`, beginning with 0, 1, ... (if nothing else is defined)
- order in the definition is important

advantage

meaningful identifiers improve readability and comprehensibility of programs

Enumeration types (cont.)

```
typedef enum
{
  Red,
  Yellow,
  Green
} Color;

int main(void)
{
  Color color;

  for (color = 0; color <= 3; color++)
    printf("color = %d\n",(int) color);
  return 0;
}
```

```
color = 0
color = 1
color = 2
color = 3
```

Data structures

Elementary datatypes

until now we only have seen elementary datatypes: `char`, `int`, `double`

Data structures ...

are used for storing logically related data

Example

store all values related to a sequence database:

- each sequence has an accession number, a creation date, a length, ...
- the sequence database consists of many sequences

Structured datatypes in C (overview)

arrays

ordered list of values of the same data type

structures

combine data of different types (all at a time)

unions

combine data of different types (one at a time)

bitfields

similar to structures, but with possibility to reduce the space requirement

Arrays

Arrays

are used for storing a fixed number of values of the same type

Areas of application:

- vectors, matrices (any dimension)
- list of names, strings,

Syntax:

Basic_datatype arrayname [number_of_elements];

Example

```
int vector[10];
short int i, b[10];
double d, s[15];
```

- number of elements must be positive and known at compile time
- if number of elements is unknown, then use dynamic memory allocation (see later sections)

Arrays in memory

- elements of arrays are stored directly one after the other in memory

4711	4712	4713 b[0]	4714 b[0]	4715 b[1]	4716 b[1]	4717 b[2]	4718 b[2]
4719 b[3]	4720 b[3]	4721 b[4]	4722 b[4]	4723 b[5]	4724 b[5]	4725 b[6]	4726 b[6]
4727 b[7]	4728 b[7]	4729 b[8]	4730 b[8]	4731 b[9]	4732 b[9]	4733	4734

Memory requirement of arrays

memory requirement

number of elements * size of basic types

Example:

- `int intarray[250]` requires $4 * 250 = 1000$ bytes
- `double d[1000]` requires $8 * 1000 = 8000$ bytes

Note

- the entire memory for an array will be allocated at runtime
- if the array is too large, this may lead to undefined behavior

Character arrays

- one of the most important applications of arrays are strings
- in C there is no datatype *String*
- instead one uses arrays over the basic type `char` or `unsigned char`
- the C-standard library contains a variety of functions for strings
- string constant: internally represented as `char`-array
- size of array is one larger than the number of characters in string constant
- additional element is used for storing the character `'\0'` at the end

Example

```
char s1[] = "Hello"; /* means the same as */
char s2[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Processing of strings

The termination character

- when processing a string, one must always check if the current character is `'\0'`
- Note:
  - if `'\0'` occurs inside of a string (which occurs e.g. if two strings are concatenated without eliminating the `'\0'`), this is an error

Initialization of Arrays

Initialization of Arrays

- global arrays are initialized with 0
- local arrays are not automatically initialized
- manual initialization in the declaration by adding a list of values

```
int vector1[5] = {6, -4, 6, 2, -1};
int vector2[5] = {6, -6, 2};
int vector3[] = {3, -5, 7, 2, -1};
char hello[5] = {'h', 'e', 'l', 'l', 'o'};
char str[] = "hello world\n";
```

- only constants are allowed in the initialization list
- there may be less elements than specified in the initialization list
- the remaining elements remain uninitialized
- implicit determination of number of elements to be allocated (see `vector3` and `str`):
  - the size of the array is defined by the length of the initialization list
  - most common use is for the initialization of string constants (as in `str`)

Access to array elements

- Syntax: *arrayname [arrayindex]*

```
int a, v[] = {1, 2, 3};
a = v[1];
```

- an array of n elements introduces n variables

Example

`int a[10]` introduces

- a variable named `a` for accessing the entire array
- it is like declaring a pointer `int *a`
- 10 variable names for the 10 array elements `a[0]`, `a[1]`, ..., `a[9]`

Index range

the first array index is always 0 and the last index is $n-1$ if the array has n elements

Access to array elements (cont.)

- the array index can be an arbitrary expression evaluating to a non-negative index
- the expression can be evaluated at runtime

```
int v1[] = {1, 2, 3}, v2[] = {4, 5, 6}, v3[3], i;
/* vector addition: v3=v1+v2 */
for (i = 0; i < 3; i++)
{
  v3[i] = v1[i] + v2[i];
}
```

Access to array elements (cont.)

access to invalid array indexes

- no check if the array index is in the allowed boundaries
- access to arbitrary memory cells
- access to values outside of array boundaries:
  - read access: program runs with incorrect values
  - write access: program write values to invalid memory cells
  - if protected memory areas are access, the program usually aborts (segmentation fault, bus error)
  - incorrect memory accesses can be detected using `valgrind`

An Array of Pointers

```
int main(void)
{
    char *ptr1 = "LaTeX";
    char *ptr2 = "MS-Word";
    char *ptr3 = "OpenOffice";
    char *arr[3]; /* an array of 3 char pointers */

    arr[0] = ptr1;
    arr[1] = ptr2;
    arr[2] = ptr3;
    printf("%s\n%s\n%s\n", arr[0],arr[1],arr[2]);
    return 0;
}
```

The output of the above program is:

```
LaTeX
MS-Word
OpenOffice
```

example from The Geek Stuff - C arrays basics explained with 13 examples

Comparison of arrays

- application of the comparison operator `==` to two arrays compares the addresses
- to check equality of arrays one compares their length and if this is equal one compares all elements

```
#include <stdbool.h>

bool array_compare(const int *a,unsigned long alen,
                   const int *b,unsigned long blen)
{
  unsigned long idx;

  if (alen != blen)
    return false;
  for (idx = 0; idx < alen; idx++)
  {
    if (a[idx] != b[idx])
      return false;
  }
  return true;
}
```


Copying of arrays

- application of the assignment operator `=` to arrays copies the addresses of two arrays
- copying an array requires to copy each element

```
void array_copy(int *a,const int *b,unsigned long blen)
{
  unsigned long idx;

  for (idx = 0; idx < blen; idx++)
  {
    a[idx] = b[idx];
  }
}
```

- Note: `a` must point to a memory area of at least 4∗`blen` bytes.

Multidimensional arrays

Applications

tables, matrices and similar structures

Syntax of definition

Basic_Datatype Name [num_of_elements1] [num_of_elements2] . . .

Example: Matrix with 2 rows and 3 columns:

```
short int v[2][3];
```

Syntax of access:

Name [arrayindex1] [arrayindex2] . . .

Example:

```
j = v[1][2];
```


Initializing multi-dimensional arrays

- in analogy to one-dimensional arrays
- rows and columns can be incompletely initialized
- size of array may not be specified

Example:

```
short int m[2][3] = {{ 2, 3, -1},
                     {-1, 4, -8}};
```

Memory layout of multi-dimensional arrays

- elements are stored row by row
- e.g. the array

```
short int m[2][3] = {{ 2, 3, -1},
                     {-1, 4, -8}};
```

is stored as follows (assuming the first element is stored at address 4713):

4711	4712	4713 m[0][0]	4714	4715 m[0][1]	4716	4717 m[0][2]	4718
4719 m[1][0]	4720	4721 m[1][1]	4722	4723 m[1][2]	4724	4725	4726

Arrays backstage

- the array variable is nothing else but a pointer
- the access to an array-element follows address-arithmetic for pointers

Example:

```
int vector[5], matrix[3][7], s;

s = vector[3];    /* equivalent to s = *(vector+3) */
s = matrix[1][2];    /* equivalent to s = *(*(matrix+1)+2) */
```

Arrays as function parameters

- when passing arrays as function parameters, the start address of the array is provided
- modifications of the array-elements are thus visible in the calling environment

Example:

in the call of `scanf` the `char`-array is passed as a pointer (i.e. the array variable is passed):

```
char str[100];
if (scanf("%s",str) == 1) ...
```

```
int i;
if (scanf("%d", &i) == 1) ...
```


Arrays as function parameters (cont.)

- to allow functions to work with arrays of different sizes, the size must usually be provided

```
void vectoradd(int *v,const int *s,const int *t,unsigned long len)
{
  unsigned long idx;
  for (idx = 0; idx < len; idx++)
    v[idx] = s[idx] + t[idx];
}
```

- alternative for `char`-arrays: end of array can be determined by scanning the string until termination character is found

```
unsigned long mystrlen(const char *s)
{
  unsigned long idx;
  for (idx = 0; s[idx] != '\0'; idx++) /* Nothing */ ;
  return idx;
}
```

Character arrays and strncpy

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

int main(void)
{
  char arr[3+1]; /* for accommodating 3 characters and '\0' byte */
  char *ptr = "abc";

  memset(arr, '\0', sizeof(arr)); /* reset all bytes */
  strncpy(arr,ptr,sizeof("abc")); /* copy string "abc" into arr */
  printf("%s\n",arr);
  arr[0] = 'p';                   /* change first character in array */
  printf("%s\n",arr);
  return 0;
}
```

example from The Geek Stuff - C arrays basics explained with 13 examples

Command line parameters

- The command line parameters are passed to the function `main`:
- complete definition: `int main(int argc, char *argv[])`:
  - `argc`: number of arguments ≥ 1
  - `argv`: pointer to array of pointers to strings: `argv[0]` is the program name, `argv[1]` is the first argument, ..., `argv[argc-1]` is the last argument

Example: the program cmd_par.c

```c
int main(int argc, char *argv[])
{
  int idx;

  printf("run program %s with %d parameters\n",argv[0], argc);
  for (idx = 0; idx < argc; idx++)
    printf("parameter %d: %s\n", idx, argv[idx]);
  return EXIT_SUCCESS;
}
```

Command line parameters (cont.)

Programm call:

```
> ./cmd_par.x foo bar
run ./cmd_par.x with 3 parameters
parameter 0: ./cmd_par.x
parameter 1: foo
parameter 2: bar
```


Application of Arrays: Searching

- Task: search a key in a sorted array of elements
- Example: does a certain name occur in the following list of names: Alfons, Beate, Ines, Klaus, Michael, Nadine, Peter, Ruth, Sabine, Sebastian, Tom?
- first approach: sequential search
  - search from first to last element, until there are no more elements or the current element is larger or equal than the key.
  - at most n comparisons are necessary for an array of n elements

Application of Arrays: Searching (cont.)

```
static long searchlinear(const char **table,unsigned long entries,
                         const char *searchkey)
{
  int cmp;
  unsigned long idx;

  for (idx = 0; idx < entries; idx++)
  {
    cmp = strcmp(searchkey,table[idx]);
    if (cmp == 0)
      return (long) idx;
    else
      if (cmp < 0)
        break;
  }
  return -1;
}
```


Application of Arrays: Searching (cont.)

```
int main(int argc,char *argv[])
{
  char searchkey[100];
  long nameindex;
  const char *names[]
    = {"Alfons", "Beate", "Ines", "Klaus", "Michael",
       "Nadine", "Peter", "Ruth", "Sabine","Sebastian", "Tom"};

  if (scanf("%s",searchkey) != 1)
  {
    fprintf(stderr,"%s: input error\n",argv[0]);
    return EXIT_FAILURE;
  }
  printf("sizeof(names)=%ld\n",sizeof(names));
  printf("sizeof(names[0])=%ld\n",sizeof(names[0]));
  nameindex = searchlinear(names,sizeof(names)/sizeof(names[0]),
      searchkey);
  if (nameindex != -1)
  {
    printf("found at position %ld\n",nameindex);
  }
  return EXIT_SUCCESS;
}
```

Application of Arrays: Searching (cont.)

Binary search is more efficient:

- start in the middle of the array
- if the key is identical to the middle element, then search is complete
- if the key is smaller than the middle element, then search in the left part of the array just before the middle element
- if the key is larger than the middle element, then search in the right part of the array just after the middle element
- iterate this until the key is found or the part to be searched is too small

at most $1 + \log_2 n$ comparisons are necessary

Application of Arrays: Searching (cont.)

```
long binarysearch(const char **table,unsigned long entries,
                  const char *searchkey)
{
  int cmp;
  const char **left = table, **mid, **right = table + entries - 1;

  while (left <= right)
  {
    mid = left + (right-left)/2;
    cmp = strcmp(searchkey,*mid);
    if (cmp == 0)
      return (long) (mid - table);
    if (cmp < 0)
      right = mid - 1;
    else
      left = mid + 1;
  }
  return -1L;
}
```

Functions for strings

function definitions in `string.h`

copy functions memcpy and memmove:

```
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```

- copies `n` bytes from `src` to `dest`
- `memcpy` requires that the memory areas pointed to by `dest` and `src` do not overlap
- can be used not only for strings, but any kind of data in memory

Functions for strings (cont.)

copy function strcpy:

```
char *strcpy(char *dest, const char *src);
```

copies strings from `src` to `dest` including the termination symbol \0

copy function strncpy:

```
char *strncpy(char *dest, const char *src, size_t n);
```

- copies at most `n` characters from `src` to `dest`
- if `src` is less than `n` characters long, the remainder of `dest` is filled with \0
- note: if `src` does not contain \0 in the first `n` symbols, then `dest` is not \0-terminated

Functions for strings (cont.)

Concatenating strings with strcat and strncat

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Comparison functions

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
```

return value:

negative	if s1 is lexicographically smaller than s2
positive	if s1 is lexicographically larger than s2
0	if s1 equal to s2

Functions for strings (cont.)

Search functions

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

return value: pointer to the first/last occurrence of `c` in `s`

Searching a set of characters in strings

```
char *strpbrk(const char *s, const char *accept);
```

return value: pointer to the first occurrence of one of the symbols from `accept` in `s`

Searching a substring

```
char *strstr(const char *haystack, const char *needle);
```

return value: pointer to the first occurrence of `needle` in `haystack`

Functions for strings (cont.)

An example using `strstr`

```
static void findallmatches(const char *haystack,const char *needle)
{
  const char *ptr;

  for (ptr = haystack; (ptr = strstr(ptr,needle)) != NULL; ptr++)
    printf("match at pos %lu\n",(unsigned long) (ptr - haystack));
}

int main(int argc,const char **argv)
{
  if (argc != 3)
  {
    fprintf(stderr,"Usage: %s <haystack> <needle>\n",argv[0]);
    return EXIT_FAILURE;
  }
  findallmatches(argv[1],argv[2]);
  return EXIT_SUCCESS;
}
```


Functions for strings (cont.)

Determining the length of strings

```
size_t strlen(const char *s);
```

return value: length of string `s`

Initializing memory area by a constant

```
void *memset(void *s, int c, size_t n);
```

fills the first `n` bytes of `s` with the byte `c`

Functions for strings (cont.)

Other functions

`strspn`	determine length of initial prefix of one string consisting of characters from other string
`strcspn`	like `strspn` but for characters *not* from other string
`strerror`	get error message string from error code
`strtok`	split string into tokens according to given separators
`memchr`	like `strchr` but for arbitrary memory areas

Function definitions in stdlib.h

convert string into integer

```
int atoi(const char *s);
long atol(const char *s);
long strtol(const char *s, char **endp, int base);
unsigned long strtoul(const char *s, char **endp, int base);
```

convert string into floating pointer number

```
double atof(const char *s);
double strtod(const char *s, char **endp);
```

Applications of arrays: distribution of random numbers

Random numbers: sequence of numbers of random values

Applications:

- cryptography (generation of keys)
- simulation
- selection of probes from large datasets

Problem:

as computers are deterministic, only pseudo random numbers can be generated with (hopefully) the following properties:

- even distribution over the range of all possible values
- numbers should be repeated as least as possible

Applications of arrays: distribution of random numbers (cont.)

Random number functions in `stdlib.h`

```
void srand48(long int seed);
double drand48(void);
```

- `srand48` initializes the random number generator with some given number (allows to make the random generation deterministic).
- `drand48` delivers random number in the interval $[0, 1)$, i.e. never 1.

Example: generate integer random numbers in the range $[1, 6]$

```
srand48(366292341L);
r = 1 + (int) (drand48()*6.0);
```

Applications of arrays: distribution of random numbers (cont.)

```
int main(int argc,char *argv[])
{
  const unsigned long trials = 100000000UL; /* 10^8 */
  const unsigned long maxnum = 6;
  unsigned long idx, randtrial, randnum, *randdist;

  srand48(366292341L); /* use some large prime number */
  randdist = calloc(maxnum+1,sizeof (*randdist));
  assert(randdist != NULL);
  for (randtrial = 0; randtrial < trials; randtrial++)
  {
    randnum = 1 + (unsigned long) (drand48() * (double) maxnum);
    randdist[randnum]++;
  }
  for (idx = 1; idx <= maxnum; idx++)
    printf("%lu %lu (%.2f)\n",idx,randdist[idx],
                              100.0 * (double) randdist[idx]/trials);
  return EXIT_SUCCESS;
}
```


Structures

A structure

combines variables of possibly different types to a single type

Syntax:

```
typedef struct
{
  type1 component1;
  type2 component2;
  ...
} Typename;
```

Structures (cont.)

Example: for a customer we need to store a name, an address and a customer-number

```
typedef struct
{
  char surname[50];
  char lastname[50];
  int number;
} Customer;

Customer customer; /* declare var. customer of type Customer */
```


Structures (cont.)

Allowed operations

- pass a structure as a parameter to a function (call by value)
- return a structure as a result of a function
- assign a structure to a variable
- get address
- `sizeof`-Operator
- element access operator

Note: in most cases structures are passed as pointers to functions and functions return pointers to structures

Example: accessing an element

```
customer.number = 1001;
printf("name: %s %s\n",customer.surname, customer.lastname);
```

Structures (cont.)

```
typedef struct {
  int day, month, year;
} Date;

typedef struct {
  char surname[50], lastname[50];
  Date first_purchase;
  int number;
} Customer_with_purchase;

  Customer_with_purchase allcustomers[100];
  allcustomers[42].first_purchase.year = 2002;
```

- the dot-operator allows access to a component of a structure
- for a pointer to a structure one can use the arrow-operator -> to access the components

Structures (cont.)

```
Date *newdatewrong(int day,int month,int year)
{ /* failure as memory of local variables is freed */
  Date date;

  date.day = day;
  date.month = month;
  date.year = year;
  return &date;
}

Date *newdate(int day,int month,int year)
{
  Date *date = malloc(sizeof(*date)); /* allocate the memory */

  assert(date != NULL);
  date->day = day;
  date->month = month;
  date->year = year;
  return date;
}
```

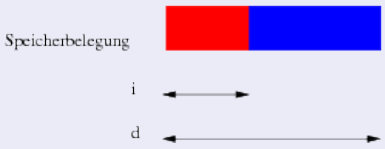
Unions

- Declaration like structures, but with the keyword `union` instead of `struct`
- only one of the specified values is stored
- memory requirement is determined by the largest element

Example

```
typedef union
{
  int i;
  double d;
} IntorDouble;
```

```
  IntorDouble u;
  u.i = 5;
  u.d = 2.3; /* u.i is
      undefined now */
```



Unions (cont.)

Applications

- save memory, if at any time only one of the components needs to be stored at once
- Unions are mostly part of a structure which contains information about which of the possible components is stored

```
typedef struct {
  unsigned char isdouble;
  union {
    int i;
    double d;
  } value;
} IntorDouble;

void inittheint(IntorDouble *id,int iv)
{
  id->isdouble = 0;
  id->value.i = iv;
}
```

Unions (cont.)

```
void initthedouble(IntorDouble *id,double dv)
{
  id->isdouble = 1;
  id->value.d = dv;
}


void showIntorDouble(IntorDouble *id)
{
  if (id->isdouble)
  {
    printf("%.2f",id->value.d);
  } else
  {
    printf("%d",id->value.i);
  }
}
```


Bitfields

- Bitfields are used to reduce the space requirement for structure components which only store a limited range of values
- specify the number of bits for a component using the `:b` notation, where `b` is an integer constant
- single bits can be specified by adding `:1` $\Rightarrow$ easy access to bits
- Syntax as in `struct`

```
typedef struct
{ /* requires 3 * 4 bytes */
  unsigned int black:1,
               value:31,
               left,
               right;
} Redblacktreenode;
```

Dynamic Memory Management

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

- `malloc` delivers a pointer to a memory area of `size` bytes
- this memory area is reserved with the call to `malloc` ⇒ dynamic memory management and allocation
- the reserved area must be freed once it is not used any more; this can be done with the function `free`
- if not freed, then the space leaks
- if `malloc` cannot deliver the requested memory area, then `NULL` is returned (this should always be checked)

Dynamic Memory Management (cont.)

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

- `realloc` adjusts the memory area pointed to by `ptr` to the size `newsize`
- thus memory area can be enlarged or reduced in size
- the contents remains unchanged in the first min$\{$`newsize`,`size`$\}$ bytes, where `size` is the current size of the memory area
- important: contents of memory area to be enlarged may be moved by `realloc`

⇒ pointers into memory may become invalid

- `realloc(NULL,size)` is equivalent to `malloc(size)`

Dynamic Memory Management (cont.)

```
static void array_reverse(long *arr, unsigned long numofentries)
{
  long *fwdptr, *bwdptr;

  for (fwdptr = arr, bwdptr = arr + numofentries - 1;
       fwdptr < bwdptr; fwdptr++, bwdptr--)
  {
    long tmp = *fwdptr;
    *fwdptr = *bwdptr;
    *bwdptr = tmp;
  }
}

static void array_show(const long *arr, unsigned long numofentries)
{
  const long *ptr;

  for (ptr = arr; ptr < arr + numofentries; ptr++)
  {
    printf("%ld\n",*ptr);
  }
}
```


Dynamic Memory Management (cont.)

```
int main(void)
{
  long value, nextfree = 0, allocated = 0, *arr = NULL;

  while (scanf("%ld",&value) == 1)
  {
    if (nextfree >= allocated)
    {
      allocated += 32L;
      arr = realloc(arr,sizeof (*arr) * allocated);
      assert(arr != NULL);
    }
    arr[nextfree++] = value;
  }
  array_reverse(arr,nextfree);
  array_show(arr,nextfree);
  free(arr);
  return EXIT_SUCCESS;
}
```

File operations

- writing to and reading from files
- in analogy to terminal I/O
- additional functions for binary files and additional low-level functions
- a file is a sequence of characters

Phases of typical file access

- open or create the file
- read or write access the file
- close the file

File operations (cont.)

Opening a file using file pointers

```
#include <stdio.h>
FILE *fopen(const char *fname,
            const char *mode);
```

- `fname`: filename (including path)
- `mode`: description of kind of access and positioning of file pointers:

mode	reading	writing	start at	comment
r	√		beginning	
w		√	beginning	file is created; delete file if exists
a		√	end	file is created; append, if exists
r+	√	√	beginning	
w+	√	√	beginning	file is created, if it does not exist
a+	√	√	end	file is created, if it does not exist

File operations (cont.)

Binary files

- append `b` to mode-string
- data will be written verbatim

Text files

- append `t` to mode-string
- data will be written after (system dependent) conversions to improve the readability by humans and the portability

Return value of fopen

- `FILE`-pointer (i.e. identifier) for the opened file
- will be used as parameter of functions for writing and reading the file

File operations (cont.)

Functions for writing and reading

- `int putc(int c, FILE *f)`
- `int getc(FILE *f):`
- `int fprintf(FILE *f, const char *format, ...)`
- `fscanf(FILE *f, const char *format, ...)`

Input functions delivers `EOF` once the end of the file is reached

Closing the file

- `int fclose(FILE *f)`
- write data from the file buffer that is not written into the file yet
- input/output is buffered to improve performance of I/O functions
- after closing, no valid access possible

File operations (cont.)

Example: Saving to a file

```
int main(int argc,char *argv[])
{
  FILE *f;

  f = fopen("test.dat", "wb");
  if (f == NULL)
  {
    fprintf(stderr,"%s: error in fopen(wb)\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  fprintf(f, "save integer 4: %d\n", 4);
  fclose(f);
  return EXIT_SUCCESS;
}
```


File operations (cont.)

Example: Reading a file

```
int main(int argc,char *argv[])
{
  FILE *f;
  int cc;

  f = fopen("test.dat", "rb");
  if (f == NULL)
  {
    fprintf(stderr,"%s: error in fopen(rb)\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  while ((cc = getc(f)) != EOF)
  {
    printf("read symbol: %c\n", cc);
  }
  printf("EOF=%d\n",EOF);
  fclose(f);
  return EXIT_SUCCESS;
}
```

Standard files

- no principal difference between output to terminal and file I/O
  - `stdin`: standard input channel
  - `stdout`: standard output channel
  - `stderr`: standard error channel
- errors should be reported using `fprintf(stderr,...)` but not `printf`

Flushing file buffers and positioning the file

- `int fflush(FILE *f)`
  - empty the file buffers
- `void rewind(FILE *F)`
  - set the current file offset to the start
- `long ftell(FILE *f)`
  - deliver the current offset of the file
- `int fseek(FILE *f, long offset, int origin)`
  - set the position to an arbitrary position
  - `offset`: distance from `origin`; can be negative
  - `origin`:
    - `SEEK_SET`: start of file
    - `SEEK_CUR`: current position
    - `SEEK_END`: end of file

Flushing file buffers and positioning the file (cont.)

Example: Determining the length of a file

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

unsigned long file_size(const char *inputfile)
{
  long filesize;
  FILE *fp = fopen(inputfile,"r");

  assert(fp != NULL);
  fseek (fp, 0, SEEK_END); /* jump to end of file */
  filesize = ftell(fp); /* read current position */
  rewind(fp); /* continue at the start of the file */
  fclose(fp);
  assert(filesize >= 0);
  return (unsigned long) filesize;
}
```


Typed files

- consider a file as a sequence of identically structured datasets
- application areas: storing of combined data types (like structures)
- alternative: store the different components of a structure one by one
  - error-prone code; difficult to maintain

Functions for types files

- opening and closing files using `fopen` and `fclose`

```
size_t fwrite(void *buf,size_t size,size_t cnt,FILE *f);
size_t fread (void *buf,size_t size,size_t cnt,FILE *f);
```

- parameter
  - `buf`: pointer to file buffer (`void *` is an untyped pointer, i.e. any kind of pointer can be passed)
  - `size`: size of a dataset
  - `cnt`: number of data sets
- return value: number of successfully written or read sets

More functions in `stdio.h`

```
int remove (const char *path);
```

remove the file or directory whose path is given

```
int unlink (const char *path);
```

remove the file whose path is given

```
int rmdir (const char *path);
```

remove the directory whose path is given

```
int rename (const char *old, const char *new);
```

rename the file with name `old` to name `new`. Both files must reside in the same file system

```
FILE *tmpfile(void);
```

create new file and return the corresponding file pointer

More functions in `stdio.h` (cont.)

```
char *fgets(char *str, int size, FILE *stream)
```

reads at most one less than the number of characters specified by `size` from the given `stream` and stores them in the string `str`. Stop at newline or at end of file or error. \0 is appended to the buffer.

```
char *gets(char *str);
```

special version of `fgets` for `stdin` and infinite size

```
int fputs(const char *str, FILE *stream);
```

write the string pointed to by `str` to the `streams`

```
int puts(const char *str);
```

special version of `fputs` for `stdout`

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

test the end-of-file indicator for the `stream`

More functions in `stdio.h` (cont.)

```
int sprintf(char *s, const char *format, ...);
```

variant of `printf` which writes to the memory area pointed to by `s`; return number of characters written

```
int snprintf(char *s, size_t size, const char *format, ...);
```

variant of `sprintf` which guarantees that no more than `size` bytes are written to `s`

Low level IO functions

- access via file descriptor (i.e. positive integer)
- file treated as an unstructured sequence of bytes which is not buffered
- the low level functions are the base for the high level functions
  - `int open(const char *path, int oflag, ...);` open the file whose `path` is given; `oflag` specifies how the file is opened (read, create, ...); if file is created, the optional third argument specifies the file permissions.
  - `ssize_t read(int fildes, void *buf, size_t nbyte);` read `nbyte` bytes from the file referenced by file descriptor `fildes` into the memory area pointed to by `buf`.
  - `ssize_t write(int fildes, const void *buf, size_t nbyte);` write `nbyte` bytes from the memory area pointed to by `buf` to the file referenced by file descriptor `fildes`.
  - `ssize_t` is used to represent the sizes of blocks that can be read or written in a single operation. Similar to `size_t`, but must be a signed type.
- application only in rare cases

Counting characters for input read from a file

```
/* A comprehensive example: distribution of characters */
#include <sys/types.h>  /* for fstat */
#include <sys/stat.h>   /* for fstat */
#include <unistd.h>     /* for fstat */
#include <fcntl.h>      /* for open */
#include <limits.h>     /* for UCHAR_MAX */
#include <ctype.h>      /* for isprint and isspace */
typedef unsigned char Uchar;

int simplefileOpen(const char *filename,unsigned long *numofbytes)
{
  int fd;
  struct stat buf;

  fd = open(filename,O_RDONLY); /* open file for read, get filedescriptor */
  if (fd == -1) /* check for error code */
  {
    fprintf(stderr,"Cannot open %s\n",filename);
    return -1;
  }
  if (fstat(fd,&buf) == -1) /* get status information of file */
  {
    fprintf(stderr,"Cannot access status of file %s\n",filename);
    return -2;
  }
  *numofbytes = buf.st_size; /* store file size in address of numofbytes */
  return fd;
}
```


Counting characters for input read from a file (cont.)

```
Uchar *readmyfile(const char *filename,unsigned long *numofbytes)
{
  Uchar *buffer;
  int fd;

  fd = simplefileOpen(filename,numofbytes); /* open file, obtain file size */
  assert(fd >= 0);
  buffer = malloc(sizeof(Uchar) * (*numofbytes)); /* space block for file */
  assert(buffer != NULL);
  if (read(fd,(void *) buffer, sizeof(Uchar) * (size_t) *numofbytes) !=
     (ssize_t) *numofbytes) /* read the given number of bytes from fd */
  {
    fprintf(stderr,"read of %ld bytes failed\n",*numofbytes);
    exit(EXIT_FAILURE);
  }
  close(fd); /* now close the descriptor, since the file has been read */
  return buffer;
}

void chardistribution(unsigned long *distribution,unsigned long numchars,
                      const Uchar *buffer,unsigned long numbytes)
{
  unsigned long idx;
  for (idx = 0; idx < numchars; idx++) /* initialize each array entry */
    distribution[idx] = 0;
  for (idx = 0; idx < numbytes; idx++) /* use ascii number of char as index */
    distribution[(int) buffer[idx]]++;
}
```

Counting characters for input read from a file (cont.)

```
void showdistribution(const unsigned long *distribution,
                      unsigned long numofchars)
{
  unsigned long idx;
  for (idx = 0; idx < numofchars; idx++)
    if (distribution[idx] > 0)
    {
      if (isprint((int) idx) && !isspace((int) idx)) /* visible char? */
        printf("%c",(char) idx);
      else
        printf("\\%lu",idx); /* show char as ascii num with prepended backsl. */
      printf("\t%8lu\n",distribution[idx]);
    }
}

int main(int argc,char *argv[])
{
  Uchar *buffer; unsigned long numofbytes, distribution[UCHAR_MAX+1];
  if (argc != 2) {
    fprintf(stderr,"Usage: %s <filename>\n",argv[0]);
    return EXIT_FAILURE;
  }
  buffer = readmyfile(argv[1],&numofbytes);
  chardistribution(distribution,UCHAR_MAX+1,buffer,numofbytes);
  showdistribution(distribution,UCHAR_MAX+1);
  free(buffer);
  return EXIT_SUCCESS;
}
```


Access to files via memory mapped IO

- the most efficient access to files is based on `mmap`
- this function delivers a pointer to a virtual memory area containing the file contents
- the corresponding address space is reserved and the file content can be processed as an array
- the part of the file which is accessed will be loaded into the main memory

$\Rightarrow$ mapping on demand

Access to files via memory mapped IO (cont.)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int simplefileOpen(const char *filename,unsigned long *numofbytes);

void *creatememorymap(const char *filename,
                      unsigned long *numofbytes)
{
  int fd;
  void *memorymap;

  fd = simplefileOpen(filename,numofbytes);
  if (fd < 0)
  {
    return NULL;
  }
  memorymap = mmap(NULL, /* no address specified for map */
                   (size_t) *numofbytes,
                   PROT_READ, /* pages may be read */
                   MAP_PRIVATE, /* unclear why need for reading */
                   fd, /* file descriptor */
                   (off_t) 0); /* offset: multiple of page size */
```


Access to files via memory mapped IO (cont.)

```
  if (memorymap == (void *) MAP_FAILED)
  {
    fprintf(stderr,"%s(%s) failed\n",__func__,filename);
    exit(EXIT_FAILURE);
  }
  return memorymap;
}

void deletememorymap(void *memorymap,unsigned long numofbytes)
{
  if (munmap(memorymap,(size_t) numofbytes) != 0)
  {
    fprintf(stderr,"%s failed\n",__func__);
    exit(EXIT_FAILURE);
  }
}
```

the main file: countchar-mmap.c:

Access to files via memory mapped IO (cont.)

```
void chardistribution(unsigned long *dist,unsigned long numofchars,
                      const Uchar *s,unsigned long numofbytes);
void showdistribution(const unsigned long *dist,
                      unsigned long numofchars);

int main(int argc,char *argv[])
{
  Uchar *s;
  unsigned long numofbytes, distribution[UCHAR_MAX+1];

  if (argc != 2)
  {
    fprintf(stderr,"Usage: %s <filename>\n",argv[0]);
    return EXIT_FAILURE;
  }
  s = creatememorymap(argv[1],&numofbytes);
  chardistribution(distribution,UCHAR_MAX+1,s,numofbytes);
  showdistribution(distribution,UCHAR_MAX+1);
  deletememorymap(s,numofbytes);
  return EXIT_SUCCESS;
}
```


Mathematical Functions in C

- header file: `math.h`
- linking option: `-lm`
- list of functions:
  - `double sin(double x)`: sinus of x
  - `double cos(double x)`: cosinus of x
  - `double tan(double x)`: tangens of x
  - `double asin(double x)`: arcussinus of $x \in [-1,1]$ in range $[-\pi/2,\pi/2]$
  - `double acos(double x)`: arcuscosinus of $x \in [-1,1]$ in range $[0,\pi]$
  - `double atan(double x)`: arcustangens of x in range $[0,\pi]$
  - `double atan2(double y, double x)`: arcustangens of y/x in range $[-\pi,\pi]$
  - `double sinh(double x)`: sinus hyperbolicus of x
  - `double cosh(double x)`: cosinus hyperbolicus of x
  - `double tanh(double x)`: tangens hyperbolicus of x

Mathematical Functions in C (cont.)

- `double exp(double x)`: exponential function, e^x
- `double log(double x)`: natural logarithm, $\ln(x)$ for $x > 0$
- `double log10(double x)`: decadic logarithm, $\log_{10}(x)$ for $x > 0$
- `double pow(double x, double y)`: power function, x^y; (error if $x = 0$ and $y \leq 0$, or $x < 0$ and y not an integer)
- `double sqrt(double x)`: square root of x, $\sqrt{x}$
- `double fabs(double x)`: absolute value of x, $|x|$
- note: `int abs(int x)` is declared in `stdlib.h`
- `double ceil(double x)`: smallest integral value $\geq$ to x: $\lceil x \rceil$
- `double floor(double x)`: largest integral value less $\leq$ to x: $\lfloor x \rfloor$

Mathematical Functions in C (cont.)

- `double ldexp (double x, int n)`: $x \cdot 2^n$
- `double frexp (double x, int *exp)`: break x into a normalized fraction in the range $[1/2, 1)$ and an integral power of 2. (which is stored in `*exp`), both 0, if $x = 0$

  example: `frexp(2560,&e)` delivers 0.625 and $e = 12$, since $0.625 \cdot 2^{12} = 2560$.
- `double modf(double x, double *ip)`

  break value x into integral (in `*ip`) and return fractional parts, each of which has the same sign as the argument
- `double fmod(double x, double y)`

  floating-point remainder of x/y.

Recursive data structures

- introduce the concepts by a larger example
- problem: count the number of occurrences of all words in a file
- solution: store the words and their counts in a binary search tree
- each node contains the following information:
  - pointer to the word, which occurs only once in the entire tree
  - counter for the number of its occurrences found yet
  - pointer to the left and right children
- each node can have up to two children; missing children are represented by `NULL`-pointers
- the words in the left (right) subtree are lexicographically smaller (larger) than the words in the considered root of the subtree

Recursive data structures (cont.)

- properties above allow to efficiently find the occurring words by traversing the paths of the tree
- if the considered subtree is empty, then the key does not occur, and one can insert a new node
- if the subtree is not empty, then consider the following cases:
  - if the key is identical to the word in the current node, then increment the counter
  - if the key is smaller (larger) than the word in the current node, then perform the insertion in the left (right) subtree

Recursive data structures (cont.)

The file `tsearch.c`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include "tsearch.h"

struct Tnode
{
  char *word;      /* points to copy of the word */
  unsigned long count;   /* number of its occurrences */
  struct Tnode *left,   /* left child */
               *right;  /* right child */
};

Tnode *tree_add(Tnode *node,const char *word)
{
  if (node == NULL)
  {
    node = malloc(sizeof *node);
    assert(node != NULL);
    node->word = strdup(word);
```


Recursive data structures (cont.)

```
    assert(node->word != NULL);
    node->count = 1UL;
    node->left = node->right = NULL;
  } else
  {
    int cond = strcmp(word,node->word);

    if (cond == 0)
    {
      node->count++;
    } else
    {
      if (cond < 0)
      {
        node->left = tree_add(node->left,word);
      } else
      {
        node->right = tree_add(node->right,word);
      }
    }
  }
  return node;
}
```

Recursive data structures (cont.)

```
void tree_print(int d,const Tnode *node)
{
  if (node != NULL)
  {
    tree_print(d+1,node->left);
    printf("%*.*s%s %lu\n",d*3,d*3,"",node->word,node->count);
    tree_print(d+1,node->right);
  }
}

void tree_delete(Tnode *node)
{
  assert(node != NULL);
  if (node->left != NULL)
    tree_delete(node->left);
  if (node->right != NULL)
    tree_delete(node->right);
  free(node->word);
  free(node);
}
```

The file `tsearch.h`

Recursive data structures (cont.)

```
#ifndef TSEARCH_H
#define TSEARCH_H

typedef struct Tnode Tnode;

Tnode *tree_add(Tnode *node,const char *word);
void tree_print(int depth,const Tnode *node);
void tree_print_tikz(int depth, const Tnode *node);
void tree_delete(Tnode *node);

#endif
```

The file `tsearch-main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tsearch.h"
#define MAXWORD 100
```

Recursive data structures (cont.)

```
int main(int argc,char *argv[])
{
  Tnode *root = NULL;
  char *ptr, word[MAXWORD+1];

  while (fgets(word,MAXWORD,stdin) != NULL)
  {
    if ((ptr = strrchr(word,'\n')) != NULL)
    {
      *ptr = '\0';
    }
    root = tree_add(root,word);
  }
  tree_print(0,root);

  if (root != NULL)
  {
    tree_delete(root);
  }
  return EXIT_SUCCESS;
}
```


Recursive data structures (cont.)

Example:
Consider the following input

```
now is the time for all good men
to come to the aid of their party
```

Then the compiled program, when reading the word list

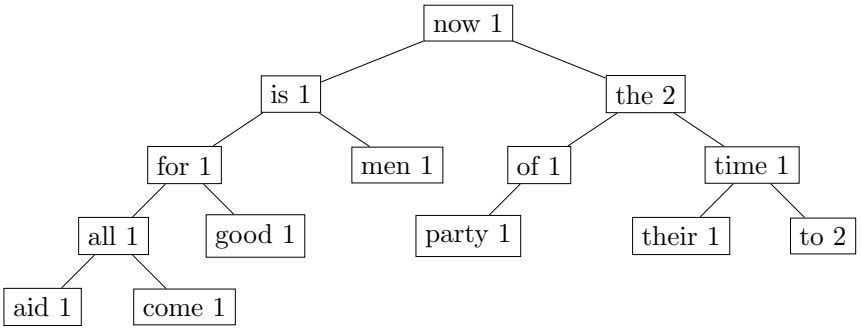
```
now
is
the
..
their
party
```

Recursive data structures (cont.)

delivers the output:

```
            aid 1
         all 1
            come 1
      for 1
         good 1
   is 1
      men 1
now 1
      of 1
         party 1
   the 2
         their 1
      time 1
         to 2
```

which represents the following binary tree:



Higher order functions

- higher order functions have other functions as parameters (which are not known at compile time)
- in C higher order functions are implemented by pointers to functions
- higher order functions form an important concept which considerably improves the extensibility and reusability of software
- Example: the function `qsort` from the standard library abstracts from the kind of data to be sorted and the order according to which the sorting is performed

```
void qsort (void *base,size_t nmemb,size_t size,
            int (*compar)(const void *,const void *));
```

Higher order functions (cont.)

- `qsort` sorts an Array with `nmemb` elements each of which has `size` bytes. `base` is the pointer to the memory area which stores the array
- the function `compar` specifies the order as a parameter to `qsort`
- the declaration of `qsort` specifies the type of `compar`
- the fourth parameter `compar` must be a function which delivers an `int`-value and has two parameters of type `const void *`. These point to the memory areas where the elements to be compared are stored
- `compar` must deliver an integer smaller than, or equal to, or larger than 0, depending on whether the first argument is smaller than, equal to, or larger than the second argument
- Example: sorting of intervals, increasing by the first component and decreasing by the second component

Higher order functions (cont.)

```
#include <stdlib.h>

typedef struct
{
  int firstindex,
      lastindex;
} Interval;

static int compareIntervals(const void *keya,const void *keyb)
{
  const Interval *ia = (const Interval *) keya;
  const Interval *ib = (const Interval *) keyb;

  if (ia->firstindex < ib->firstindex)
  {
    return -1;
  }
  if (ia->firstindex > ib->firstindex)
  {
    return 1;
  }
```

Higher order functions (cont.)

```
  if (ia->lastindex < ib->lastindex)
  {
    return 1;
  }
  if (ia->lastindex > ib->lastindex)
  {
    return -1;
  }
  return 0;
}

void sortintervals(Interval *intervaltab,size_t numofintervals)
{
  qsort(intervaltab,numofintervals,sizeof *intervaltab,
        compareIntervals);
}
```


Higher order functions (cont.)

– pointer to functions do not need to be explicitly dereferenced. One can use them like ordinary functions.

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

typedef int Vectorbasetype;

typedef struct
{
  unsigned int vlength;
  Vectorbasetype *vvalues;
} Vector;

typedef Vectorbasetype(*Vectorop)(Vectorbasetype,Vectorbasetype);
```

Higher order functions (cont.)

```
void applyvectorop(Vector *vresult,const Vector *v1,
                   const Vector *v2, Vectorop vop)
{
  unsigned int i;
  if (v1->vlength != v2->vlength)
  {
    fprintf(stderr,"vectors must have equal length\n");
    exit(EXIT_FAILURE);
  }
  vresult->vlength = v1->vlength;
  vresult->vvalues = malloc(sizeof(Vectorbasetype) *
                            vresult->vlength);
  if (vresult->vvalues == NULL)
  {
    fprintf(stderr,"Cannot allocate space for result vector\n");
    exit(EXIT_FAILURE);
  }
  for (i = 0; i < vresult->vlength; i++)
  {
    vresult->vvalues[i] = vop(v1->vvalues[i],v2->vvalues[i]);
  }
}
```


Higher order functions (cont.)

```
Vectorbasetype addvalues(Vectorbasetype value1,
                         Vectorbasetype value2)
{
  return value1 + value2;
}

Vectorbasetype diffvalues(Vectorbasetype value1,
                          Vectorbasetype value2)
{
  return value1 - value2;
}

void vectoradd(Vector *vresult,const Vector *v1,
               const Vector *v2) {
  applyvectorop(vresult,v1,v2,addvalues);
}

void vectordiff(Vector *vresult,const Vector *v1,
                const Vector *v2) {
  applyvectorop(vresult,v1,v2,diffvalues);
}
```

Threads for concurrent programming

Threaded programming

- provides software developers with a useful abstraction of concurrent execution of statements.
- the threads of the program execute in parallel, so that multithreaded programs can operate faster on computer systems that have CPUs with multiple cores
- applicable if the problem can be divided into independent subproblems
- threads may share data for
  - reading (no problem) and
  - writing (problem if two threads want to manipulate the same memory cell $\Rightarrow$ need mechanisms that lock such a memory cell)

`phtreads` provide a widely accepted implementation of threads
Example shows a program which computes a sum over a range of indices using threads.

Threads for concurrent programming (cont.)

```
#include <pthread.h>

/* A range specifies a continues range of indices, for which
   we want to compute the sum. Each range has its own sum. */

typedef struct
{
  unsigned long lower, upper, sum;
} Range;

/* We first split the range from 0 to totalwidth into parts
   of approximately the same width according to the parameter
   parts. */

static Range *ranges_new(unsigned long totalwidth,
                         unsigned int parts)
{
  Range *ranges = malloc(sizeof (*ranges) * parts);
  unsigned long width;
  unsigned int idx;

  assert(ranges != NULL);
  width = totalwidth/parts;
```

Threads for concurrent programming (cont.)

```
  for (idx = 0; idx < parts; idx++)
  {
    ranges[idx].lower
      = (idx == 0) ? 0
                   : ranges[idx-1].upper + 1;
    ranges[idx].upper
      = (idx < parts - 1) ? (idx+1) * width - 1
                          : totalwidth - 1;
  }
  return ranges;
}

/* The following function takes a range and computes the sum of
   the values in the range. This could of course be done by some
   simple arithmetic. In real application one e.g. has an array
   of numbers and these have to be summed in the given range.
   The function can be executed by a thread, as the ranges of
   different threads are disjoint. */

static void *sumofrange (void *arg)
{
  unsigned long idx;
  Range *r = (Range *) arg;
```


Threads for concurrent programming (cont.)

```
  r->sum = 0;
  for (idx = r->lower; idx <= r->upper; idx++)
    r->sum += idx;
  return NULL;
}

/* create parts threads, each executing sumofrange for a range. */

static pthread_t *threads_start(Range *ranges,unsigned int parts)
{
  unsigned int idx;
  pthread_t *threadtab = malloc(sizeof (*threadtab) * parts);
  assert(threadtab != NULL);
  for (idx = 0; idx < parts; idx++)
  {
    printf("run thread %u for range %lu %lu\n",
            idx,ranges[idx].lower,ranges[idx].upper);
    pthread_create (&threadtab[idx], NULL, sumofrange, ranges+idx);
    /* The main program continues while the thread executes. */
  }
  return threadtab;
}
```

Threads for concurrent programming (cont.)

```
/* For each part, suspend executing the calling thread (which is
   the main thread) until the corresponding thread is terminating.
   So the main thread waits for the worker threads. Once a thread
   has terminated, the sum value is available and the partial sums
   are summed to give the complete sum. After the for-loop only
   the main thread is running. */

static unsigned long threads_join(pthread_t *threadtab,
                                  const Range *ranges,
                                  unsigned int parts)

{
  unsigned int idx;
  unsigned long sum = 0;

  for (idx = 0; idx < parts; idx++)
  {
    pthread_join(threadtab[idx], NULL);
    sum += ranges[idx].sum;
  }
  return sum;
}
```


Threads for concurrent programming (cont.)

```
#define USAGE fprintf(stderr,"Usage: %s <width> <parts>\n",\
                      argv[0]);\
              exit(EXIT_FAILURE)

/* The program runs in five phases:
   - parsing of the two parameters, i.e. the totalwidth and
     the number of parts
   - creation of ranges
   - creation of the threads
   - joining of the threads
   - cleaning the data structures
*/

int main (int argc,char *argv[])
{
  pthread_t *threadtab;
  Range *ranges;
  unsigned long sum;
  long totalwidth;
  int parts;

  if (argc != 3)
```

Threads for concurrent programming (cont.)

```
  {
    USAGE;
  }
  if (sscanf(argv[1],"%ld",&totalwidth) != 1 || totalwidth <= 0)
  {
    USAGE;
  }
  if (sscanf(argv[2],"%d",&parts) != 1 || parts <= 0)
  {
    USAGE;
  }
  ranges = ranges_new(totalwidth,parts);
  threadtab = threads_start(ranges,parts);
  sum = threads_join(threadtab,ranges,parts);
  printf("sum = %lu\n",sum);
  free(threadtab);
  free(ranges);
  return EXIT_SUCCESS;
}
```