Linux-Kommandozeilenreferenz von fosswire.com

Unix/Linux Command Reference

FOSSwire.com

File Commands

ls – directory listing
ls -al – formatted listing with hidden files
cd *dir* - change directory to *dir*
cd – change to home
pwd – show current directory
mkdir *dir* – create a directory *dir*
rm *file* – delete *file*
rm -r *dir* – delete directory *dir*
rm -f *file* – force remove *file*
rm -rf *dir* – force remove directory *dir* *
cp *file1 file2* – copy *file1* to *file2*
cp -r *dir1 dir2* – copy *dir1* to *dir2*; create *dir2* if it doesn't exist
mv *file1 file2* – rename or move *file1* to *file2* if *file2* is an existing directory, moves *file1* into directory *file2*
ln -s *file link* – create symbolic link *link* to *file*
touch *file* – create or update *file*
cat > *file* – places standard input into *file*
more *file* – output the contents of *file*
head *file* – output the first 10 lines of *file*
tail *file* – output the last 10 lines of *file*
tail -f *file* – output the contents of *file* as it grows, starting with the last 10 lines

Process Management

ps – display your currently active processes
top – display all running processes
kill *pid* – kill process id *pid*
killall *proc* – kill all processes named *proc* *
bg – lists stopped or background jobs; resume a stopped job in the background
fg – brings the most recent job to foreground
fg *n* – brings job *n* to the foreground

File Permissions

chmod *octal file* – change the permissions of *file* to *octal*, which can be found separately for user, group, and world by adding:

- 4 – read (r)
- 2 – write (w)
- 1 – execute (x)

Examples:
chmod 777 – read, write, execute for all
chmod 755 – rwx for owner, rx for group and world
For more options, see **man chmod**.

SSH

ssh *user@host* – connect to *host* as *user*
ssh -p *port user@host* – connect to *host* on port *port* as *user*
ssh-copy-id *user@host* – add your key to *host* for *user* to enable a keyed or passwordless login

Searching

grep *pattern files* – search for *pattern* in *files*
grep -r *pattern dir* – search recursively for *pattern* in *dir*
command* | grep *pattern – search for *pattern* in the output of *command*
locate *file* – find all instances of *file*

System Info

date – show the current date and time
cal – show this month's calendar
uptime – show current uptime
w – display who is online
whoami – who you are logged in as
finger *user* – display information about *user*
uname -a – show kernel information
cat /proc/cpuinfo – cpu information
cat /proc/meminfo – memory information
man *command* – show the manual for *command*
df – show disk usage
du – show directory space usage
free – show memory and swap usage
whereis *app* – show possible locations of *app*
which *app* – show which *app* will be run by default

Compression

tar cf *file.tar files* – create a tar named *file.tar* containing *files*
tar xf *file.tar* – extract the files from *file.tar*
tar czf *file.tar.gz files* – create a tar with Gzip compression
tar xzf *file.tar.gz* – extract a tar using Gzip
tar cjf *file.tar.bz2* – create a tar with Bzip2 compression
tar xjf *file.tar.bz2* – extract a tar using Bzip2
gzip *file* – compresses *file* and renames it to *file.gz*
gzip -d *file.gz* – decompresses *file.gz* back to *file*

Network

ping *host* – ping *host* and output results
whois *domain* – get whois information for *domain*
dig *domain* – get DNS information for *domain*
dig -x *host* – reverse lookup *host*
wget *file* – download *file*
wget -c *file* – continue a stopped download

Installation

Install from source:
./configure
make
make install
dpkg -i *pkg.deb* – install a package (Debian)
rpm -Uvh *pkg.rpm* – install a package (RPM)

Shortcuts

Ctrl+C – halts the current command
Ctrl+Z – stops the current command, resume with **fg** in the foreground or **bg** in the background
Ctrl+D – log out of current session, similar to **exit**
Ctrl+W – erases one word in the current line
Ctrl+U – erases the whole line
Ctrl+R – type to bring up a recent command
!! - repeats the last command
exit – log out of current session

* use with extreme caution.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

Number Literals

Integers

`0b11111111`	binary	`0B11111111`	binary
`0377`	octal	`255`	decimal
`0xff`	hexadecimal	`0xFF`	hexadecimal

Real Numbers

`88.0f` / `88.1234567f`

single precision float (`f` suffix)

`88.0` / `88.123456789012345`

double precision float (no `f` suffix)

Signage

`42` / `+42`	positive	`-42`	negative

Binary notation `0b...` / `0B...` is available on GCC and most but not all C compilers.

Variables

Declaring

`int x;`	A variable.
`char x = 'C';`	A variable & initialising it.
`float x, y, z;`	Multiple variables of the same type.
`const int x = 88;`	A constant variable: can't assign to after declaration (compiler enforced.)

Naming

`johnny5IsAlive;` ✔	Alphanumeric, not a keyword, begins with a letter.
~~`2001`~~`ASpaceOddysey;` ✖	Doesn't begin with a letter.
~~`while`~~`;` ✖	Reserved keyword.
`how`~~`-exciting!`~~`;` ✖	Non-alphanumeric.
~~`iamaverylongvariablenameohmygoshyesiam`~~`;` ✖	Longer than 31 characters (C89 & C90 only)

Constants are `CAPITALISED`. Function names usually take the form of a verb eg. `plotRobotUprising()`.

Primitive Variable Types

**applicable but not limited to most ARM, AVR, x86 & x64 installations*

`[class] [qualifier] [unsigned] type/void name;`

by ascending arithmetic conversion

Integers

Type	*Bytes*	*Value Range*
`char`	1	`unsigned` **OR** `signed`
`unsigned char`	1	0 to 2^8-1
`signed char`	1	-2^7 to 2^7-1
`int`	2 / 4	`unsigned` **OR** `signed`
`unsigned int`	2 / 4	0 to 2^{16}-1 **OR** 2^{31}-1
`signed int`	2 / 4	-2^{15} to 2^{15}-1 **OR** -2^{31} to 2^{32}-1
`short`	2	`unsigned` **OR** `signed`
`unsigned short`	2	0 to 2^{16}-1
`signed short`	2	-2^{15} to 2^{15}-1
`long`	4 / 8	`unsigned` **OR** `signed`
`unsigned long`	4 / 8	0 to 2^{32}-1 **OR** 2^{64}-1
`signed long`	4 / 8	-2^{31} to 2^{31}-1 **OR** -2^{63} to 2^{63}-1
`long long`	8	`unsigned` **OR** `signed`
`unsigned long long`	8	0 to 2^{64}-1
`signed long long`	8	-2^{63} to 2^{63}-1

Floats

Type	*Bytes*	*Value Range (Normalized)*

By **Ashlyn Black**
cheatography.com/ashlyn-black/

Published 28th January, 2015.
Last updated 20th April, 2015.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

Primitive Variable Types (cont)

`float`	4	$\pm 1.2\times10^{-38}$ to $\pm 3.4\times10^{38}$
`double`	8 / 4	$\pm 2.3\times10^{-308}$ to $\pm 1.7\times10^{308}$ **OR** alias to `float` for AVR.
`long double`	ARM: 8, AVR: 4, x86: 10, x64: 16	

Qualifiers

`const type`	Flags variable as read-only (compiler can optimise.)
`volatile type`	Flags variable as unpredictable (compiler cannot optimise.)

Storage Classes

`register`	Quick access required. May be stored in RAM**OR** a register. Maximum size is register size.
`static`	Retained when out of scope. `static` global variables are confined to the scope of the compiled object file they were declared in.
`extern`	Variable is declared by another file.

Typecasting

`(type)a`	Returns `a` as data `type`.

Primitive Variable Types (cont)

```
char x = 1, y = 2; float z = (float) x / y;
```

Some types (denoted with **OR**) are architecture dependant.

There is no primitive boolean type, only zero (false, `0`) and non-zero (true, usually `1`.)

Extended Variable Types

```
[class] [qualifier] type name;
```

by ascending arithmetic conversion

From the `stdint.h` Library

Type	*Bytes*	*Value Range*
`int8_t`	1	-2^7 to 2^7-1
`uint8_t`	1	0 to 2^8-1
`int16_t`	2	-2^{15} to $2^{15}-1$
`uint16_t`	2	0 to $2^{16}-1$
`int32_t`	4	-2^{31} to $2^{31}-1$
`uint32_t`	4	0 to $2^{32}-1$
`int64_t`	8	-2^{63} to $2^{63}-1$
`uint64_t`	8	0 to $2^{64}-1$

From the `stdbool.h` Library

Type	*Bytes*	*Value Range*
`bool`	1	`true` / `false` or `0` / `1`

The `stdint.h` library was introduced in C99 to give integer types architecture-independent lengths.

Structures

Defining

`struct strctName{ type x; type y; };`	A structure type `strctName` with two members, `x` and `y`. *Note trailing semicolon*
`struct item{ struct item *next; };`	A structure with a recursive structure pointer inside. Useful for linked lists.

Declaring

`struct strctName varName;`	A variable `varName` as structure type `strctName`.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

Structures (cont)

`struct strctName *ptrName;`	A `strctName` structure type pointer, `ptrName`.
`struct strctName{ type a; type b; } varName;`	Shorthand for defining `strctName` and declaring `varName` as that structure type.
`struct strctName varName = { a, b };`	A variable `varName` as structure type `strctName` and initialising its members.

Accessing

`varName.x`	Member `x` of structure `varName`.
`ptrName->x`	Value of structure pointer `ptrName` member `x`.

Bit Fields

`struct{char a:4, b:4} x;`	Declares `x` with two members `a` and `b`, both four bits in size (0 to 15.)

Array members can't be assigned bit fields.

Type Definitions

Defining

`typedef unsigned short uint16;`	Abbreviating a longer type name to `uint16`.
`typedef struct structName{int a, b;}newType;`	Creating a `newType` from a structure.
`typedef enum typeName{false, true}bool;`	Creating an enumerated `bool` type.

Declaring

`uint16 x = 65535;`	Variable `x` as type `uint16`.
`newType y = {0, 0};`	Structure `y` as type `newType`.

Unions

Defining

`union uName{int x; char y[8];}`	A union type `uName` with two members, `x` & `y`. Size is same as biggest member size.

Declaring

`union uN vName;`	A variable `vName` as union type `uN`.

Accessing

`vName.y[int]`	Members cannot store values concurrently. Setting `y` will corrupt `x`.

Unions are used for storing multiple data types in the same area of memory.

Enumeration

Defining

`enum bool { false, true };`	A custom data type `bool` with two possible states: `false` or `true`.

Declaring

`enum bool varName;`	A variable `varName` of data type `bool`.

Assigning

`varName = true;`	Variable `varName` can only be assigned values of either `false` or `true`.

Evaluating

`if(varName == false)`	Testing the value of `varName`.

Pointers

Declaring

`type *x;`	Pointers have a data `type` like normal variables.

By **Ashlyn Black**
cheatography.com/ashlyn-black/

Published 28th January, 2015.
Last updated 20th April, 2015.

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

Pointers (cont)

`void *v;`	They can also have an incomplete type. Operators other than assignment cannot be applied as the length of the type is unknown.
`struct type *y;`	A data structure pointer.
`type z[];`	An array/string name can be used as a pointer to the first array element.

Accessing

`x`	A memory address.
`*x`	Value stored at that address.
`y->a`	Value stored in structure pointer `y` member `a`.
`&varName`	Memory address of normal variable `varName`.
`*(type *)v`	Dereferencing a `void` pointer as a `type` pointer.

A pointer is a variable that holds a memory location.

Arrays

Declaring

`type name[int];`	You set array length.
`type name[int] = {x, y, z};`	You set array length and initialise elements.
`type name[int] = {x};`	You set array length and initialise all elements to `x`.
`type name[] = {x, y, z};`	Compiler sets array length based on initial elements.

Size cannot be changed after declaration.

Dimensions

`name[int]`	One dimension array.
`name[int][int]`	Two dimensional array.

Accessing

`name[int]`	Value of element `int` in array `name`.

Arrays (cont)

`*(name + int)`	Same as `name[int]`.

Elements are contiguously numbered ascending from `0`.

`&name[int]`	Memory address of element `int` in array `name`.
`name + int`	Same as `&name[int]`.

Elements are stored in contiguous memory.

Measuring

`sizeof(array) / sizeof(arrayType)`	Returns length of `array`. *(Unsafe)*
`sizeof(array) / sizeof(array[0])`	Returns length of `array`. *(Safe)*

Strings

`'A'` character	Single quotes.
`"AB"` string	Double quotes.
`\0`	Null terminator.

Strings are `char` arrays.

```
char name[4] = "Ash";
```

is equivalent to

```
char name[4] = {'A', 's', 'h', '\0'};
```

```
int i; for(i = 0; name[i]; i++){}
```

`\0` *evaluates as false.*

Strings must include a `char` element for `\0`.

Escape Characters

`\a`	alarm (bell/beep)	`\b`	backspace
`\f`	formfeed	`\n`	newline
`\r`	carriage return	`\t`	horizontal tab
`\v`	vertical tab	`\\`	backslash
`\'`	single quote	`\"`	double quote
`\?`	question mark		
`\nnn`	Any octal ANSI character code.		
`\xhh`	Any hexadecimal ANSI character code.		

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

Functions

Declaring

```
type/void funcName([args...]){ [return var;] }
```

*Function names follow the same restrictions as variable names but must **also** be unique.*

`type/void`	Return value type (`void` if none.)
`funcName()`	Function name and argument parenthesis.
`args...`	Argument types & names (`void` if none.)
`{}`	Function content delimiters.
`return var;`	Value to return to function call origin. Skip for `void` type functions. Functions exit immediately after a `return`.

By Value vs By Pointer

`void f(type x); f(y);`	Passing variable `y` to function `f` argument `x` (by value.)
`void f(type *x); f(array);`	Passing an array/string to function `f` argument `x` (by pointer.)
`void f(type *x); f(structure);`	Passing a structure to function `f` argument `x` (by pointer.)
`void f(type *x); f(&y);`	Passing variable `y` to function `f` argument `x` (by pointer.)
`type f(){ return x; }`	Returning by value.
`type f(){ type x; return &x; }`	Returning a variable by pointer.

Functions (cont)

`type f(){ static type x[]; return &x; }`	Returning an array/string/structure by pointer. The `static` qualifier is necessary otherwise `x` won't exist after the function exits.

Passing by pointer allows you to change the originating variable within the function.

Scope

```
int f(){ int i = 0; } i++; ✖
```

`i` is declared inside `f()`, it doesn't exist outside that function.

Prototyping

```
type funcName(args...);
```

Place before declaring or referencing respective function (usually before `main`.)

`type funcName([args...])`	Same `type`, `name` and `args...` as respective function.
`;`	Semicolon instead of function delimiters.

main()

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

Anatomy

`int main`	Program entry point.
`int argc`	# of command line arguments.
`char *argv[]`	Command line arguments in an array of strings. #1 is always the program filename.
`return int;`	Exit status (`integer`) returned to the OS upon program exit.

Command Line Arguments

`app two 3`	Three arguments, `"app"`, `"two"` and `"3"`.
`app "two 3"`	Two arguments, `"app"` and `"two 3"`.

`main` is the first function called when the program executes.

By **Ashlyn Black**
cheatography.com/ashlyn-black/

Published 28th January, 2015.
Last updated 20th April, 2015.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

Conditional (Branching)

if, else if, else

`if(a) b;`	Evaluates `b` if `a` is true.
`if(a){ b; c; }`	Evaluates `b` and `c` if `a` is true.
`if(a){ b; }else{ c; }`	Evaluates `b` if `a` is true, `c` otherwise.
`if(a){ b; }else if(c){ d; }else{ e; }`	Evaluates `b` if `a` is true, otherwise `d` if `c` is true, otherwise `e`.

switch, case, break

`switch(a){ case b: c; }`	Evaluates `c` if `a` equals `b`.
`switch(a){ default: b; }`	Evaluates `b` if `a` matches no other case.
`switch(a){ case b: case c: d; }`	Evaluates `d` if `a` equals either `b` or `c`.
`switch(a){ case b: c; case d: e; default: f; }`	Evaluates `c`, `e` and `f` if `a` equals `b`, `e` and `f` if `a` equals `d`, otherwise `f`.
`switch(a){ case b: c; break; case d: e; break; default: f; }`	Evaluates `c` if `a` equals `b`, `e` if `a` equals `d` and `e` otherwise.

Iterative (Looping)

while

```
int x = 0; while(x < 10){ x += 2; }
```

Loop skipped if test condition initially false.

`int x = 0;`	Declare and initialise integer `x`.
`while()`	Loop keyword and condition parenthesis.
`x < 10`	Test condition.
`{}`	Loop delimiters.
`x += 2;`	Loop contents.

do while

```
char c = 'A'; do { c++; } while(c != 'Z');
```

Always runs through loop at least once.

`char c = 'A';`	Declare and initialise character `c`.

Iterative (Looping) (cont)

`do`	Loop keyword.
`{}`	Loop delimiters.
`c++;`	Loop contents.
`while();`	Loop keyword and condition parenthesis. *Note semicolon.*
`c != 'Z'`	Test condition.

for

```
int i; for(i = 0; n[i] != '\0'; i++){} (C89)
```

OR

```
for(int i = 0; n[i] != '\0'; i++){} (C99+)
```

Compact increment/decrement based loop.

`int i;`	Declares integer `i`.
`for()`	Loop keyword.
`i = 0;`	Initialises integer `i`. *Semicolon.*
`n[i] != '\0';`	Test condition. *Semicolon.*
`i++`	Increments `i`. *No semicolon.*
`{}`	Loop delimiters.

continue

```
int i=0; while(i<10){ i++; continue; i--;}
```

Skips rest of loop contents and restarts at the beginning of the loop.

break

```
int i=0; while(1){ if(x==10){break;} i++; }
```

Skips rest of loop contents and exits loop.

Console Input/Output

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

Characters

`getchar()`	Returns a single character's ANSI code from the input stream buffer as an *integer*. *(safe)*
`putchar(int)`	Prints a single character from an ANSI code *integer* to the output stream buffer.

Strings

Console Input/Output (cont)

`gets(strName)`	Reads a line from the input stream into a string variable. *(Unsafe, removed in C11.)*
Alternative	
`fgets(strName, length, stdin);`	Reads a line from the input stream into a string variable. *(Safe)*
`puts("string")`	Prints a string to the output stream.

Formatted Data

`scanf("%d", &x)`	Read value/s (type defined by format string) into variable/s (type must match) from the input stream. Stops reading at the first whitespace. `&` *prefix not required for arrays (including strings.) (unsafe)*
`printf("I love %c %d!", 'C', 99)`	Prints data (formats defined by the format string) as a string to the output stream.
Alternative	
`fgets(strName, length, stdin); sscanf(strName, "%d", &x);`	Uses `fgets` to limit the input length, then uses `sscanf` to read the resulting string in place of `scanf`. *(safe)*

The stream buffers must be flushed to reflect changes. String terminator characters can flush the output while newline characters can flush the input.

Safe functions are those that let you specify the length of the input. *Unsafe* functions do not, and carry the risk of memory overflow.

File Input/Output

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

Opening

```
FILE *fptr = fopen(filename, mode);
```

`FILE *fptr`	Declares `fptr` as a FILE type pointer (stores stream location instead of memory location.)
`fopen()`	Returns a stream location pointer if successful,`0` otherwise.
`filename`	String containing file's directory path & name.
`mode`	String specifying the file access mode.
Modes	
`"r"` / `"rb"`	Read existing text/binary file.
`"w"` / `"wb"`	Write new/over existing text/binary file.
`"a"` / `"ab"`	Write new/append to existing text/binary file.
`"r+"` / `"r+b"` / `"rb+"`	Read and write existing text/binary file.
`"w+"` / `"w+b"` / `"wb+"`	Read and write new/over existing text/binary file.
`"a+"` / `"a+b"` / `"ab+"`	Read and write new/append to existing text/binary file.

Closing

`fclose(fptr);`	Flushes buffers and closes stream. Returns `0` if successful, `EOF` otherwise.

Random Access

`ftell(fptr)`	Return current file position as a long integer.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

File Input/Output (cont)

`fseek(fptr, offset, origin);`	Sets current file position. Returns *false* is successful, *true* otherwise. The `offset` is a long integer type.
Origins	
`SEEK_SET`	Beginning of file.
`SEEK_CUR`	Current position in file.
`SEEK_END`	End of file.

Utilities

`feof(fptr)`	Tests end-of-file indicator.
`rename(strOldName, strNewName)`	Renames a file.
`remove(strName)`	Deletes a file.

Characters

`fgetc(fptr)`	Returns character read or EOF if unsuccessful. *(safe)*
`fputc(int c, fptr)`	Returns character written or EOF if unsuccessful.

Strings

`fgets(char *s, int n, fptr)`	Reads `n-1` characters from file `fptr` into string `s`. Stops at `EOF` and `\n`. *(safe)*
`fputs(char *s, fptr)`	Writes string `s` to file `fptr`. Returns non-negative on success, `EOF` otherwise.

Formatted Data

`fscanf(fptr, format, [...])`	Same as `scanf` with additional file pointer parameter. *(unsafe)*
`fprintf(fptr, format, [...])`	Same as `printf` with additional file pointer parameter.
Alternative	

File Input/Output (cont)

`fgets(strName, length, fptr);` `sscanf(strName, "%d", &x);`	Uses `fgets` to limit the input length, then uses `sscanf` to read the resulting string in place of `scanf`. *(safe)*

Binary

`fread(void *ptr, sizeof(element), number, fptr)`	Reads a `number` of `element`s from `fptr` to array `*ptr`. *(safe)*
`fwrite(void *ptr, sizeof(element), number, fptr)`	Writes a `number` of `element`s to file `fptr` from array `*ptr`.

Safe functions are those that let you specify the length of the input. *Unsafe* functions do not, and carry the risk of memory overflow.

Placeholder Types (f/printf And f/scanf)

`printf("%d%d...", arg1, arg2...);`

Type	Example	Description
`%d` or `%i`	`-42`	Signed decimal integer.
`%u`	`42`	Unsigned decimal integer.
`%o`	`52`	Unsigned octal integer.
`%x` or `%X`	`2a or 2A`	Unsigned hexadecimal integer.
`%f` or `%F`	`1.21`	Signed decimal float.
`%e` or `%E`	`1.21e+9` or `1.21E+9`	Signed decimal w/ scientific notation.
`%g` or `%G`	`1.21e+9` or `1.21E+9`	Shortest representation of `%f`/`%F` or `%e`/`%E`.
`%a` or `%A`	`0x1.207c8ap+30` or `0X1.207C8AP+30`	Signed hexadecimal float.
`%c`	`a`	A character.
`%s`	`A String.`	A character string.

By **Ashlyn Black**
cheatography.com/ashlyn-black/

Published 28th January, 2015.
Last updated 20th April, 2015.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

Placeholder Types (f/printf And f/scanf) (cont)

%p	A pointer.
%% %	A percent character.
%n	No output, saves # of characters printed so far. Respective printf argument must be an integer pointer.

The pointer format is architecture and implementation dependant.

Placeholder Formatting (f/printf And f/scanf)

%[Flags][Width][.Precision][Length]Type

Flags

-	Left justify instead of default right justify.
+	Sign for both positive numbers and negative.
#	Precede with 0, 0x or 0X for %o, %x and %X tokens.
space	Left pad with spaces.
0	Left pad with zeroes.

Width

integer	Minimum number of characters to print: invokes padding if necessary. Will not truncate.
*	Width specified by a preceding argument in printf.

Precision

.integer	Minimum # of digits to print for %d, %i, %o, %u, %x, %X. Left pads with zeroes. Will not truncate. Skips values of 0.
	Minimum # of digits to print after decimal point for %a, %A, %e, %E, %f, %F (default of 6.)
	Minimum # of significant digits to print for %g & %G.
	Maximum # of characters to print from %s (a string.)
.	If no integer is given, default of 0.

Placeholder Formatting (f/printf And f/scanf) (cont)

.*	Precision specified by a preceding argument in printf.

Length

hh	Display a char as int.
h	Display a short as int.
l	Display a long integer.
ll	Display a long long integer.
L	Display a long double float.
z	Display a size_t integer.
j	Display a intmax_t integer.
t	Display a ptrdiff_t integer.

Preprocessor Directives

#include <inbuilt.h>	Replaces line with contents of a standard C header file.
#include "./custom.h"	Replaces line with contents of a custom header file. *Note dir path prefix & quotations.*
#define NAME value	Replaces all occurrences of NAME with value.

Comments

```
// We're single-line comments!
// Nothing compiled after // on these lines.
/* I'm a multi-line comment!
   Nothing compiled between
   these delimiters. */
```

C Reserved Keywords

_Alignas	break	float	signed
_Alignof	case	for	sizeof
_Atomic	char	goto	static
_Bool	const	if	struct
_Complex	continue	inline	switch
_Generic	default	int	typedef
_Imaginary	do	long	union

By **Ashlyn Black**
cheatography.com/ashlyn-black/

Published 28th January, 2015.
Last updated 20th April, 2015.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

C Reserved Keywords (cont)

_Noreturn	double	register	unsigned
_Static_assert	else	restrict	void
_Thread_local	enum	return	volatile
auto	extern	short	while
_A-Z...	__...		

C / POSIX Reserved Keywords

E[0-9]...	E[A-Z]...	is[a-z]...	to[a-z]...
LC_[A-Z]...	SIG[A-Z]...	SIG_[A-Z]...	str[a-z]...
mem[a-z]...	wcs[a-z]...	..._t	

GNU Reserved Names

Header Reserved Keywords

Name	*Reserved By Library*
d_...	dirent.h
l_...	fcntl.h
F_...	fcntl.h
O_...	fcntl.h
S_...	fcntl.h
gr_...	grp.h
..._MAX	limits.h
pw_...	pwd.h
sa_...	signal.h
SA_...	signal.h
st_...	sys/stat.h
S_...	sys/stat.h
tms_...	sys/times.h
c_...	termios.h
V...	termios.h
I...	termios.h
O...	termios.h
TC...	termios.h
B[0-9]...	termios.h

GNU Reserved Names

Heap Space

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

Allocating

`malloc();`	Returns a memory location if successful, `NULL` otherwise.
`type *x; x = malloc(sizeof(type));`	Memory for a variable.
`type *y; y = malloc(sizeof(type) * length );`	Memory for an array/string.
`struct type *z; z = malloc(sizeof(struct type));`	Memory for a structure.

Deallocating

`free(ptrName);`	Removes the memory allocated to `ptrName`.

Reallocating

`realloc(ptrName, size);`	Attempts to resize the memory block assigned to `ptrName`.

The memory addresses you see are from virtual memory the operating system assigns to the program; they are not physical addresses.

Referencing memory that isn't assigned to the program will produce an OS segmentation fault.

The Standard Library

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

Randomicity

`rand()`	Returns a (predictable) random integer between 0 and RAND_MAX based on the randomiser seed.
`RAND_MAX`	The maximum value `rand()` can generate.
`srand(unsigned integer);`	Seeds the randomiser with a positive integer.
`(unsigned) time(NULL)`	Returns the computer's tick-tock value. Updates every second.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

The Standard Library (cont)

Sorting

```
qsort(array, length, sizeof(type), compFunc);
```

`qsort()`	Sort using the QuickSort algorithm.
`array`	Array/string name.
`length`	Length of the array/string.
`sizeof(type)`	Byte size of each element.
`compFunc`	Comparison function name.

compFunc

```
int compFunc( const void *a, const void b* ){ return( *(int *)a - *(int *)b); }
```

`int compFunc()`	Function name unimportant but must return an integer.
`const void *a, const void *b`	Argument names unimportant but must identical otherwise.
`return( *(int *)a - *(int *)b);`	Negative result swaps `b` for `a`, positive result swaps `a` for `b`, a result of `0` doesn't swap.

C's inbuilt randomiser is cryptographically insecure: DO NOT use it for security applications.

The Character Type Library

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

`tolower(char)`	Lowercase `char`.
`toupper(char)`	Uppercase `char`.
`isalpha(char)`	True if `char` is a letter of the alphabet, false otherwise.
`islower(char)`	True if `char` is a lowercase letter of the alphabet, false otherwise.
`isupper(char)`	True if `char` is an uppercase letter of the alphabet, false otherwise.
`isnumber(char)`	True if `char` is numerical (`0` to `9`) and false otherwise.

The Character Type Library (cont)

`isblank`	True if `char` is a whitespace character (`' '`, `'\t'`, `'\n'`) and false otherwise.

The String Library

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

`strlen(a)`	Returns # of `char` in string `a` as an integer. Excludes `\0`. *(unsafe)*
`strcpy(a, b)`	Copies strings. Copies string `b` over string `a` up to and including `\0`. *(unsafe)*
`strcat(a, b)`	Concatenates strings. Copies string `b` over string `a` up to and including `\0`, starting at the position of `\0` in string `a`. *(unsafe)*
`strcmp(a, b)`	Compares strings. Returns *false* if string `a` equals string `b`, *true* otherwise. Ignores characters after `\0`. *(unsafe)*
`strstr(a, b)`	Searches for string `b` inside string `a`. Returns a pointer if successful, `NULL` otherwise. *(unsafe)*

Alternatives

`strncpy(a, b, n)`	Copies strings. Copies `n` characters from string `b` over string `a` up to and including `\0`. *(safe)*
`strncat(a, b, n)`	Concatenates strings. Copies `n` characters from string `b` over string `a` up to and including `\0`, starting at the position of `\0` in string `a`. *(safe)*
`strncmp(a, b, n)`	Compares first `n` characters of two strings. Returns *false* if string `a` equals string `b`, *true* otherwise. Ignores characters after `\0`. *(safe)*

Safe functions are those that let you specify the length of the input. *Unsafe* functions do not, and carry the risk of memory overflow.

Cheatography

C Reference Cheat Sheet

by Ashlyn Black via cheatography.com/20410/cs/3196/

The Time Library

```
#include <time.h>
```

Variable Types

`time_t`	Stores the calendar time.
`struct tm *x;`	Stores a time & date breakdown.
tm structure members:	
`int tm_sec`	Seconds, 0 to 59.
`int tm_min`	Minutes, 0 to 59.
`int tm_hour`	Hours, 0 to 23.
`int tm_mday`	Day of the month, 1 to 31.
`int tm_mon`	Month, 0 to 11.
`int tm_year`	Years since 1900.
`int tm_wday`	Day of the week, 0 to 6.
`int tm_yday`	Day of the year, 0 to 365.
`int tm_isdst`	Daylight saving time.

Functions

`time(NULL)`	Returns unix epoch time (seconds since 1/Jan/1970.)
`time(&time_t);`	Stores the current time in a `time_t` variable.
`ctime(&time_t)`	Returns a `time_t` variable as a string.
`x = localtime( &time_t);`	Breaks `time_t` down into `struct tm` members.

Unary Operators

by descending evaluation precedence

`+a`	Sum of `0` (zero) and `a`. (0 + a)
`-a`	Difference of `0` (zero) and `a`. (0 - a)
`!a`	Complement (logical NOT) of `a`. (~a)
`~a`	Binary ones complement (bitwise NOT) of `a`. (~a)
`++a`	Increment of `a` by `1`. (a = a + 1)
`--a`	Decrement of `a` by `1`. (a = a - 1)
`a++`	Returns `a` then increments `a` by `1`. (a = a + 1)

Unary Operators (cont)

`a--`	Returns `a` then decrements `a` by `1`. (a = a - 1)
`(type)a`	Typecasts `a` as `type`.
`&a;`	Memory location of `a`.
`sizeof(a)`	Memory size of `a` (or `type`) in bytes.

Binary Operators

by descending evaluation precedence

`a * b;`	Product of `a` and `b`. (a × b)
`a / b;`	Quotient of dividend `a` and divisor `b`. Ensure divisor is non-zero. (a ÷ b)
`a % b;`	Remainder of *integers* dividend `a` and divisor `b`.
`a + b;`	Sum of `a` and `b`.
`a - b;`	Difference of `a` and `b`.
`a << b;`	Left bitwise shift of `a` by `b` places. ($a \times 2^b$)
`a >> b;`	Right bitwise shift of `a` by `b` places. ($a \times 2^b$)
`a < b;`	Less than. True if `a` is less than `b` and false otherwise.
`a <= b;`	Less than or equal to. True if `a` is less than or equal to `b` and false otherwise. ($a \leq b$)
`a > b;`	Greater than. True if `a` is greater than than `b` and false otherwise.
`a >= b;`	Greater than or equal to. True if `a` is greater than or equal to `b` and false otherwise. ($a \geq b$)
`a == b;`	Equality. True if `a` is equal to `b` and false otherwise. ($a \Leftrightarrow b$)
`a != b;`	Inequality. True if `a` is not equal to `b` and false otherwise. ($a \neq b$)
`a & b;`	Bitwise AND of `a` and `b`. ($a \cap b$)
`a ^ b;`	Bitwise exclusive-OR of `a` and `b`. ($a \oplus b$)

By **Ashlyn Black**
cheatography.com/ashlyn-black/

Published 28th January, 2015.
Last updated 20th April, 2015.

Binary Operators (cont)

`a | b;`	Bitwise inclusive-OR of `a` and `b`. (a ⋃ b)
`a && b;`	Logical AND. True if both `a` and `b` are non-zero. (Logical AND) (a ⋂ b)
`a || b;`	Logical OR. True if either `a` or `b` are non-zero. (Logical OR) (a ⋃ b)

Ternary & Assignment Operators

by descending evaluation precedence

`x ? a : b;`	Evaluates `a` if `x` evaluates as true or `b` otherwise. (if(x){ a; } else { b; })
`x = a;`	Assigns value of `a` to `x`.
`a *= b;`	Assigns product of `a` and `b` to `a`. (a = a × b)
`a /= b;`	Assigns quotient of dividend `a` and divisor `b` to `a`. (a = a ÷ b)
`a %= b;`	Assigns remainder of *integers* dividend `a` and divisor `b` to `a`. (a = a mod b)
`a += b;`	Assigns sum of `a` and `b` to `a`. (a = a + b)
`a -= b;`	Assigns difference of `a` and `b` to `a`. (a = a - b)
`a <<= b;`	Assigns left bitwise shift of `a` by `b` places to `a`. (a = a × 2^b)
`a >>= b;`	Assigns right bitwise shift of `a` by `b` places to `a`. (a = a × 2^{-b})
`a &= b;`	Assigns bitwise AND of `a` and `b` to `a`. (a = a ⋂ b)
`a ^= b;`	Assigns bitwise exclusive-OR of `a` and `b` to `a`. (a = a ⊕ b)
`a |= b;`	Assigns bitwise inclusive-OR of `a` and `b` to `a`. (a = a ⋃ b)

C Cheatsheet by Ashlyn Black

ashlynblack.com

By **Ashlyn Black**
cheatography.com/ashlyn-black/

Published 28th January, 2015.
Last updated 20th April, 2015.

Kurzeinführung in Linux

im Rahmen des Programmierkurses

Programmieren in C 2015

Editor

Um ein Programm zu schreiben, wird ein ***Editor*** benötigt. Mit ihm wird eine Quelltextdatei erzeugt, in die der Quelltext in einer für Menschen lesbaren Form geschrieben und anschließend auf der Festplatte gespeichert wird. Die Quelltextdatei kann der Computer nicht ausführen. Der Quelltext muss dafür in ***Maschinensprache*** – eine Folge von Nullen und Einsen – übersetzt werden. Das macht der ***Compiler***.

Viele Text-Editoren stehen unter Linux zur Verfügung. Hier wird der Editor ***kate*** verwendet. Er hat den Vorteil, dass in ihm das sogenannte ***Terminal*** mit Kommandozeileninterpreter geöffnet werden kann. Zwar bieten auch andere Editoren diese Möglichkeit, aber meist nicht derart komfortabel. Letztendlich ist es jedoch auch Geschmacksache, welchen Editor der Nutzer bevorzugt. Wem es mehr liegt, Short-Cuts zu benutzen, der kann *kate* und ein Terminal so öffnen, dass sie sich auf dem Bildschirm direkt nebeneinander befinden, um dann mit Ctrl+TAB zwischen ihnen zu wechseln.

Das aufklappbare Hauptmenü des ***K Desktop Enviroment (KDE)*** von OpenSuse enthält am oberen Rand eine Suchfunktion, in deren Textzeile *kate* eingegeben und aufgerufen werden kann. Alternativ kann der Befehl *kate* einfach in ein Terminal eingetippt und mit *Return* ausgeführt werden. Achtung, Linux unterscheidet zwischen Groß- und Kleinschreibung! Die KDE mit aufgeklapptem Hauptmenü ist in Abbildung 3 zu sehen. Sie ist der Arbeitsumgebung von Windows ähnlich.

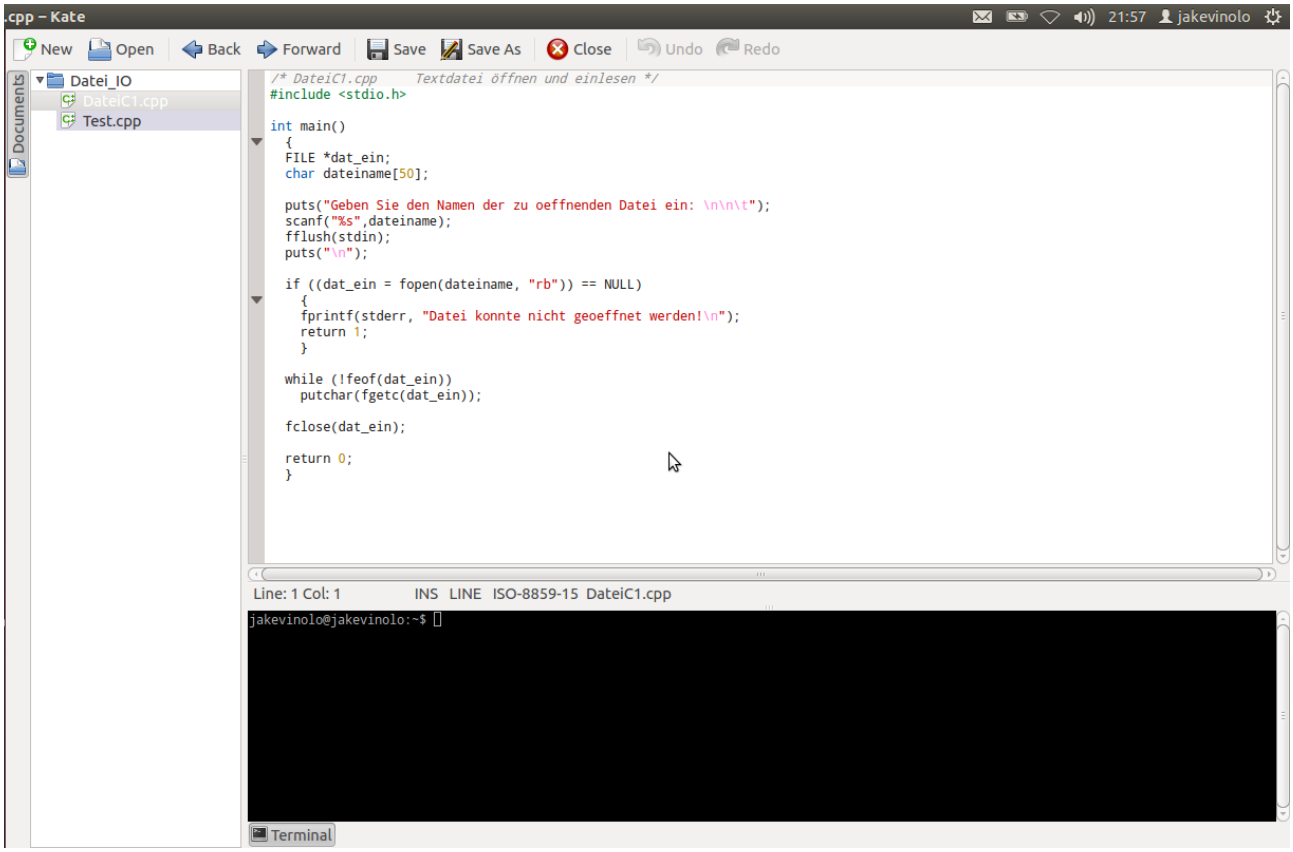

Abbildung 1: *Der Editor **kate** bietet neben dem Eingabefenster für den Quelltext auch das Terminal an.*

Terminal

Das Terminal ist der für Linux typische ***Kommandozeileninterpreter***. Dieser stellt die Schnittstelle zwischen Benutzer und Kernel dar. Der Kernel ist der Systemkern (Software), der zum Beispiel die Kommunikation mit der Hardware eines Rechners übernimmt. Abbildung 2 zeigt ein solches Terminal, wie es unter OpenSuse mit der KDE in der Rubrik ***Favoriten*** im ***aufklappbaren Hauptmenü*** zu finden ist. Das aufklappbare Hauptmenü zeigt Abbildung 3. Das Terminal gehört zur sogenannten ***Shell***, die den Kernel umgibt. In jede Shell ist eine eigene Skriptsprache eingebettet, die benutzt werden kann, um Prozesse zu automatisieren. Im Terminal können zum Beispiel Ordnerinhalte angesehen oder der Compiler aufgerufen werden, um eine Quelltextdatei in Maschinencode zu übersetzen. Da in C programmiert werden soll, muss als Compiler der ***GNU C Compiler:*** `gcc` (http://gcc.gnu.org/) verwendet werden.

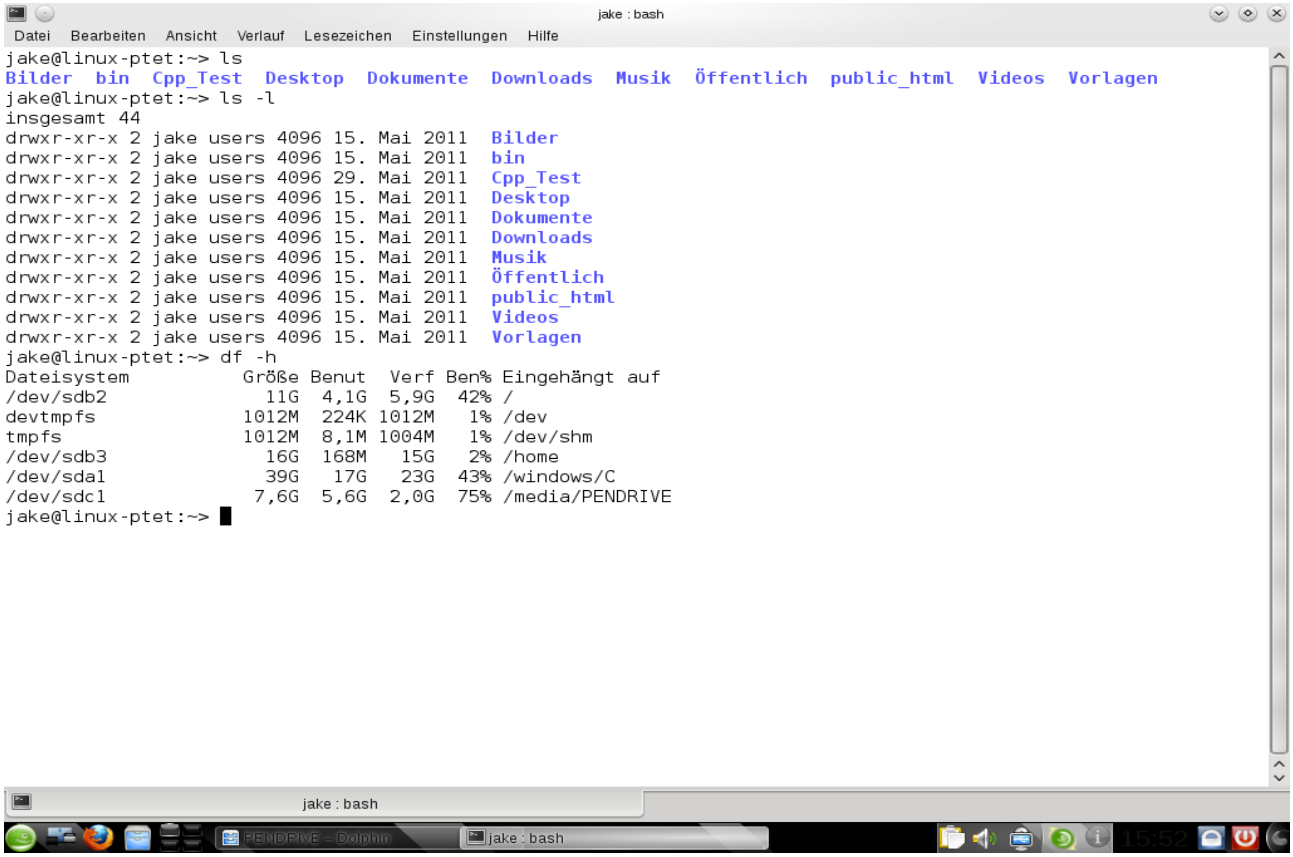

Abbildung 2: *Der für Linux typische Kommandozeileninterpreter (Terminal) enthält eine eigene Skriptsprache.*

Quelltext und Compiler

Ein Programm zu schreiben und es ausführbar zu machen, bedeutet damit zunächst,

1. den Quelltext in einen Editor einzugeben, z.B.: ***kate***,
2. den Quelltext als eine Datei mit der Endung .c, z.B.: **`test.c`**, in einem Ordner abzuspeichern,
3. den GNU C Compiler `gcc` in der Kommandozeile in diesem Ordner aufzurufen und als Parameter den Namen der Quelltextdatei zu übergeben, dann mit Return auszuführen:
   `jake@linux-ptet:~>` **`gcc test.c`**

Die ausführbare Datei befindet sich dann im selben Ordner und hat den Namen a.out.
Sie lässt sich ausführen, indem in die Kommandozeile "`jake@linux-ptet:~>` **`./a.out`**" eingegeben und mit Return bestätigt wird.

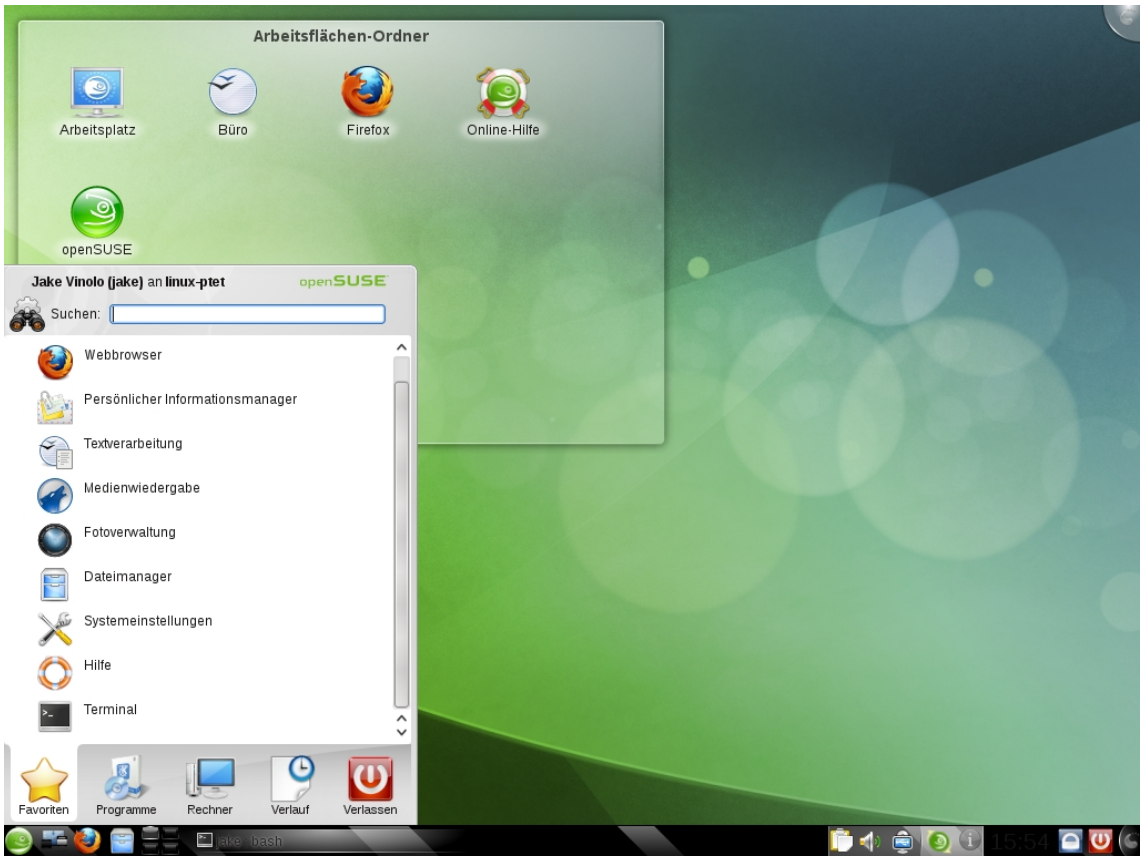

Abbildung 3: *Das sogenannte K Desktop Enviroment (KDE) bietet eine ähnliche Arbeitsumgebung wie unter Windows. Zu sehen ist links das aufgeklappte Hautmenü.*

Verzeichnisbaum unter Linux

Das für den Anfang wichtigste Verzeichnis (= Ordner) ist das ***home***-Verzeichnis eines Benutzers. Wenn sich der Benutzer mit seinem *benutzernamen* und seinem *passwort* an einem Linux-Rechner anmeldet, dann wird sein Profil (= persönliche Einstellungen des Systems) aus dem entsprechenden Benutzerverzeichnis des home-Verzeichnisses geladen. Über dem home-Verzeichnis, neben dem es noch eine Reihe anderer gibt, befindet sich das sogenannte ***Wurzelverzeichnis*** mit dem Namen "*/*". Die Abbildungen 4 und 5 zeigen den gesamten Verzeichnisbaum. In der Abbildung 5 ist das home-Verzeichnis mit den Benutzer-Verzeichnissen "*hans*" und "*susi*" zu sehen. "hans" und "susi" sind die Benutzerverzeichnisse, die zu den Benutzernamen "*hans*" und "*susi*" gehören.

Terminalbefehle für die Verzeichnisverwaltung

In Abbildung 2 ist ein Terminal zu sehen, das aufgerufen wurde, nachdem sich der Benutzer angemeldet hat. Hinter dem Eingabeprompt "`jake@linux-ptet:~>`" in der ersten Zeile steht der Befehl **`ls`** (= list). Wird er mit Return ausgeführt, zeigt er alle Verzeichnisse im Benutzerverzeichnis "*jake*" an.

Der Befehl **`ls -l`** gibt detaillierte Informationen zu den Ordnern an. Dort stehen der Dateityp, die Zugriffsrechte, Anzahl der Hardlinks zu Dateien, der Eigentümer, die Gruppenzugehörigkeit, die Dateigröße in Byte, Änderungsdatum und Uhrzeit sowie der Dateiname.
Siehe auch unter http://wiki.ubuntuusers.de/ls.

Mit "**`cd Dokumente`**"wird in das Unterverzeichnis *Dokumente* gewechselt. Zurück in das übergeordnete Verzeichnis führt der Befehl "**`cd ..`**". Der Befehl "**`exit`**" schließt das Terminal.

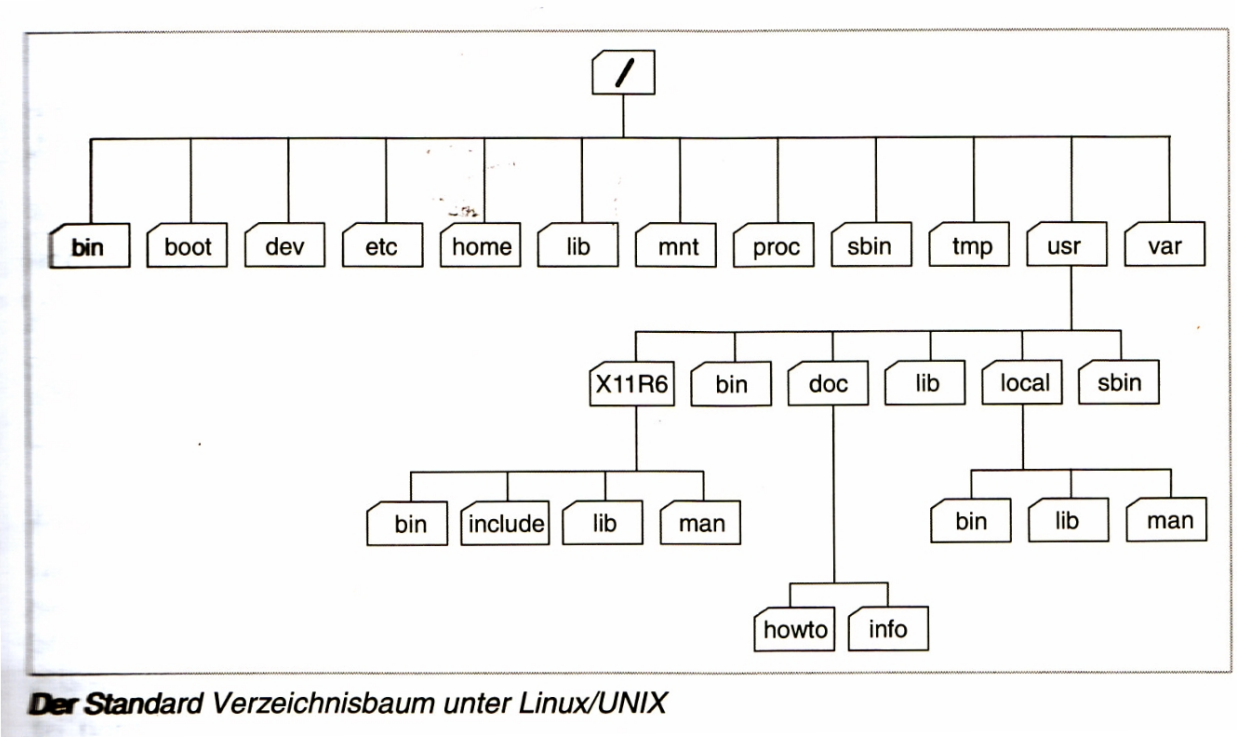

Abbildung 4: *Der für Linux typische Verzeichnisbaum mit dem home-Verzeichnis, in dem die Benutzerverzeichnisse liegen.*

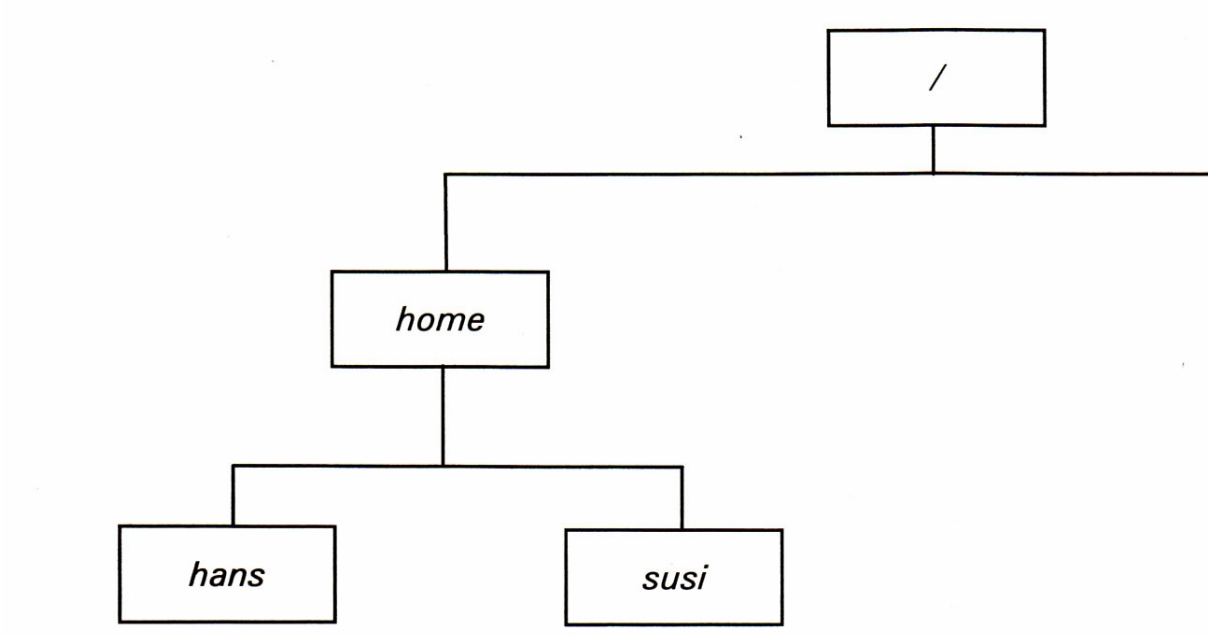

Abbildung 5: *Unter dem Wurzelverzeichnis "/" befindet sich das home-Verzeichnis mit den Benutzerverzeichnissen.*

Erreichbarkeit der Rechner im Physikalischen Rechnerpool

Die Rechner des Physikalischen Rechnerpools sind per SSH erreichbar. Damit ist es möglich, sich mit der Linux-Umgebung von OpenSuse vertraut zu machen.

Sie können sich auch von außerhalb des Universitätsnetzes mit Ihrem Konto des Physikalischen Rechnerpools an einem der dortigen Rechner anmelden, indem Sie folgende Befehlszeile verwenden:

```
ssh -X nutzername@prpXX.prp.physik.tu-darmstadt.de
```

Das Kommando `ssh` stellt eine verschlüsselte Verbindung zu dem Rechner `prpXX` der Domäne `prp.physik.tu-darmstadt.de` her. Jeder Rechner hat eine Nummer XX, die Sie vor Ort nachschauen können.
Mit der Option `-X` wird das sogenannte *X11-Forwarding* eingeschaltet. Damit lassen sich grafische Programme, die per SSH auf dem Rechner gestartet werden, auf dem eigenen Bildschirm anzeigen.

Quellen

[1] RRZN Universität Hannover: *Linux- Nutzung mit der grafischen Oberfläche KDE 2*, 2. Auflage, Hannover, April 2001.

[2] RRZN Universität Hannover: *Programmierung – Grundlagen – mit Beispielen in Java*, 6. Auflage, Hannover, September 2008.

[3] http://www.fosswire.com, Abfrage 28.7.2014.

[4] RRZN Universität Hannover: *C – Die Programmiersprache C. Ein Nachschlagewerk*. 12. Auflage, Hannover, April 2001.

[5] RRZN Universität Hannover: Unix – Eine Einführung in die Benutzung. 15. Auflage, Hannover, November 2001.

[6] H. Herold, J. Arndt: *C-Programmierung unter Linux*. SuSE Press (2001).