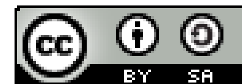
Portabler Code

Allgemeine Betrachtung und Techniken

Portabilität zwischen OS / Compilern

Portabilität zwischen Hardwareplattformen

Werkzeuge & Tipps



Harald Nikolisin

Portabilität

verringert Abhängigkeiten von Herstellern/Produkten

ermöglicht Erschließung neuer Märkte

kann Alleinstellungsmerkmal sein

bedient spezialisierte Nischen zu akzeptablen Kosten

bietet mehr Möglichkeiten der Entwicklung

erhöht die Qualität der Software

erhöht die Kosten der Softwareerstellung

ist oftmals ein schlichtes Muss

portierbarem Code

vs.

Portierung von Code

Vorbeugen ist der Heilung vorzuziehen

Portierung (nachträglich) schwieriger als der portable Entwurf

Regelmäßige Checks unerlässlich

Richtiges Maß finden abseits von Dogmen

So portabel wie sinnvoll, aber nicht mehr

Wahl der Arbeitsmittel

Welche Programmiersprachen kommen in Betracht?

Welche Bibliotheken werden eingesetzt?

Mit welchen Werkzeugen wird gearbeitet?

Das Gelingen portabler Software hängt entscheidend von der Wahl der Programmiersprache und der Bibliotheken ab.

Eine Akzeptanz kann nur erreicht werden, wenn der Mehraufwand für den Entwickler überschaubar und leicht handzuhaben ist.

Die Werkzeuge hingegen müssen nur Mindestanforderungen genügen.

Arbeitsmittel: Programmiersprachen I

Anforderungen:

Muss dem Einsatzzweck genügen (Performance,Features)
Muss an die in Betracht kommenden Bibliotheken ankoppelbar sein
Muss international standardisiert sein
Werkzeuge müssen auf den möglichen Zielplattformen vorhanden sein
Sollte eine aktive Weiterentwicklung haben

C wurde ursprünglich entwickelt um UNIX von einer bestimmten Hardwarearchitektur zu entkoppeln.

Beste Voraussetzungen bieten folgende Sprachen
(jeweils mit dem letzten verfügbaren Standardisierung)

C	ISO/IEC 9899/1999 (C99)
C++	ISO/IEC 14882:2011 (C++11)
Fortran	ISO/IEC 1539-1:2010 (Fortran 2008)

Arbeitsmittel: Programmiersprache II

was gibt es sonst noch?

Skriptsprachen:

oftmals gutes Featureset / Bindings
hoch portabel
überall verfügbar
Performance nicht für fat-clients ausreichend
kein Standard, nur Referenzimplementation

Java / C#

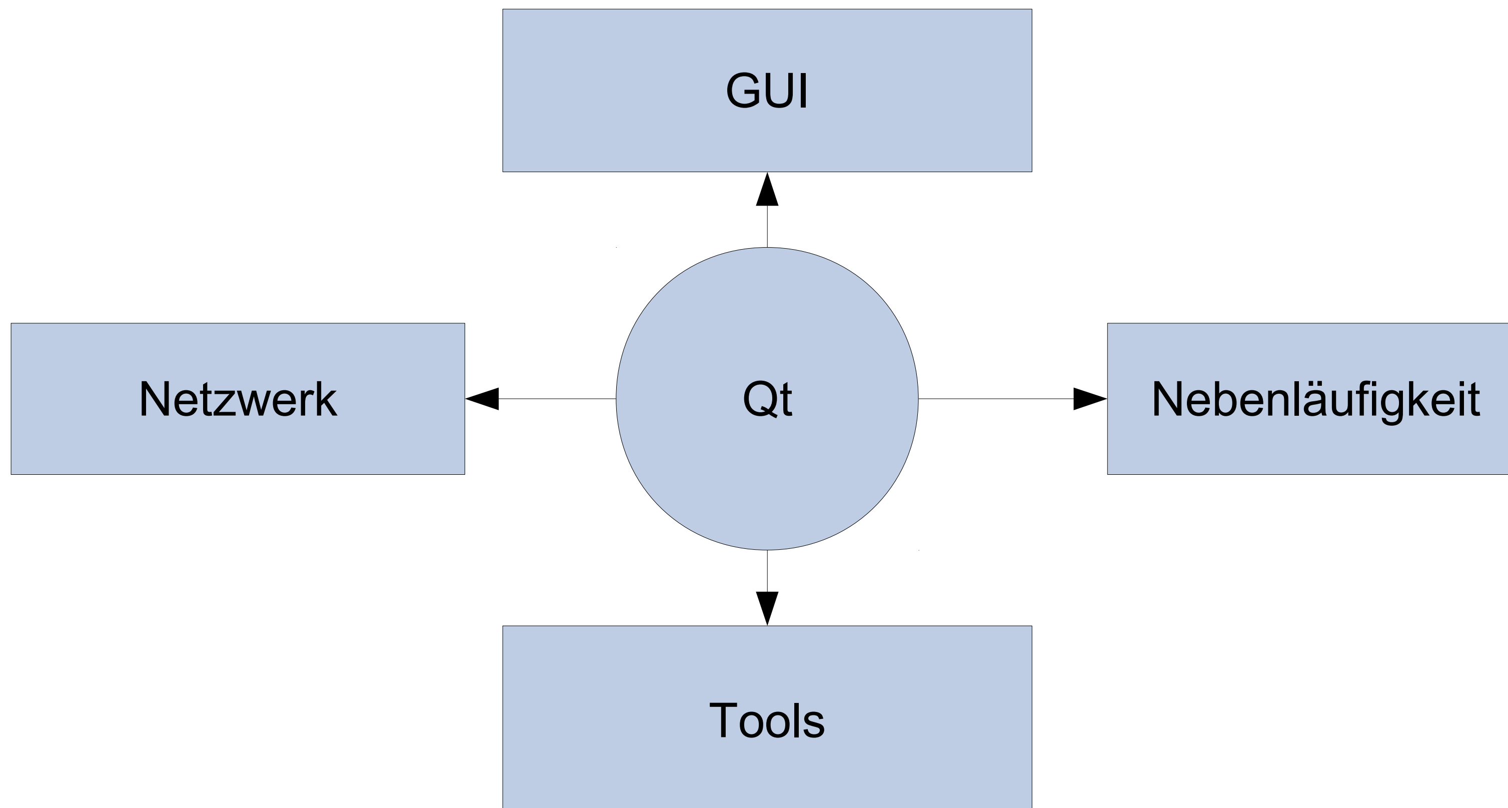
sehr gutes Featureset
C# standardisiert / Java unter GPL/Community Process
Aktive Weiterentwicklung hängt zu stark nur von einer Firma ab
schlechtere Performance
ausreichende Hilfsmittel auf allen Plattformen nicht gesichert

Assembler

Hochperformant
extrem abhängig von der Hardwareplattform

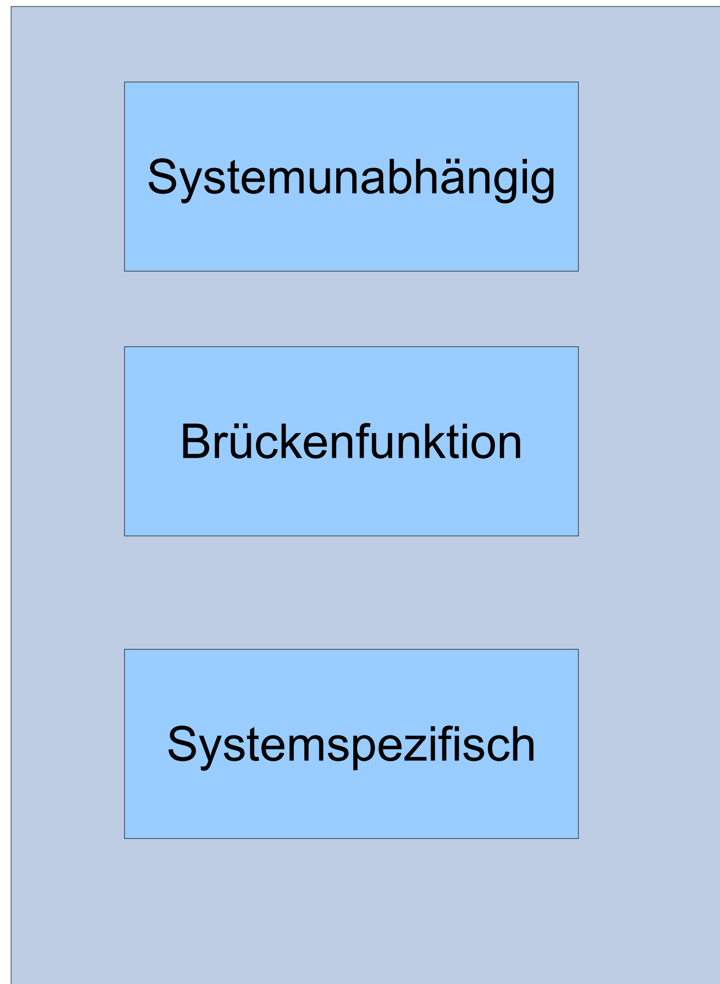
Arbeitsmittel: Bibliotheken

Die Bibliotheken tragen die Hauptlast der Portabilität. Sie müssen dem Anwendungsentwickler ein plattformneutrales API zur Verfügung stellen, bei dem sich der Programmierer keine Sorgen um die Implementation machen muss.



weitere gute universelle (plattformneutrale) API's: zlib,libpng,lapack..

Brückenfunktionen, Zwischenschicht



Trennung in systemabhängige und systemunabhängige Teile, zwischen denen Brückenfunktionen eine Verbindung herstellen.

Plattformabhängige files trennen:

`processmanagement_win32.c`
`processmanagement_linux.c`

Bezeichner in C / globale Bezeichner sollten eindeutig sein
bspw. *Reverse Internet Naming*
`de.company.XXXX`

Lange Bezeichner sollten kein Problem mehr darstellen
(32 signifikante Stellen ab C99)

Generelle Hinweise

klare Bezeichner, Mnemonic beachten

englischsprachige Kommentare

so wenig Annahmen wie möglich, v.a. bei GUI Programmierung

Kapselung ist einer ad-hoc Trennung (DEFINES) immer vorzuziehen

typedef ist sicherer als #define

Übersicht Betriebssysteme

non-GUI		GUI		
Windows	UNIX	Windows	X11 (UNIX,Linux)	Aqua (MAC OS X)

Fortran / C / C++

WinAPI	POSIX	.net	xlib	Cocoa

clib / fclib — (GKS)

Qt

Plattformkennungen

Umgebungsvariablen setzen oder vom System verwenden

Kann durch die Build Umgebung gesetzt (Skripte)

Werte können sein *WIN32*,*LINUX*,*MACOSX*

Die Compilerscripts setzen diese Definition direkt(-D$SYSTEM)

zentralen Header schreiben, der bspw. POSIX Gemeinsamkeiten definiert (#define UNIX = Linux, MAC OS X, Oracle Solaris)

Die **Qt** Bibliothek definiert

Q_WS_WIN

Q_WS_X11

Q_WS_MACX

Für Fortran eigenen Präprozessor schreiben oder Compilerspezifischen nehmen (verringert die Portabilität)

Oftmals sind Compiler- anstatt Betriebssystem Bedingungen gemeint

_MSC_VER

__GNUC__

__MINGW

Die *Q_WS_* Definitionen sollte nur in Programmteilen verwendet werden, die direkt Window-Manager betreffende Funktionalität enthält.

OS Unterschiede: non-GUI

Windows		UNIX
CRLF (0D0A)	**Zeilenende**	LF (0A)
local 8bit (z.B. Latin1 aka Windows CP-1252)	**Standard-Textkodierung**	UTF-8
%TEST%	**Umgebungsvariablen**	$TEST
;	**Variablentrenner**	:
\	**Pfadtrenner**	/
<LAUFWERKSBUCHSTABE>:\ UNC Laufwerksname	**Root Verzeichnis**	/

das Rad muß nicht neu erfunden werden

Features der Qt Bibliothek

Prozesshandling

Threadhandling

Pluginhandling

Speicherung Benutzerdaten

Pfad / Dateioperationen

Alignment Binärdatei

I/O Handling

...

für fehlende Dinge (global Defines) eigene Library schreiben

Libraries without hell I

Folgende Konzepte sind auf allen Plattformen gleichermaßen vorhanden.

statische Bibliothek (Archiv)

dynamische Bibliothek

Plugin (optionale Bibliotheksroutinen)

DELAYLOAD Feature nur unter Windows -> besser Plugin verwenden

Unter Windows müssen Funktionen explizit exportiert werden, realisiert mit:

```
#ifndef MYLIB_TYP
#ifdef Q_WS_WIN
#   define MYLIB __declspec(dllimport)
#else
#   define MYLIB
#endif
#endif
```

Libraries without hell II

Suchpfade zur Laufzeit[1]

Windows

1. (nur .net: Manifest Resources)
2. Anwendungsverzeichnis
3. Windows Systemverzeichnis
4. Windows Verzeichnis
5. %PATH%
6. working Directory

Linux

1. DT_RPATH (ohne DT_RUNPATH)[2]
2. $LD_LIBRARY_PATH[3]
3. DT_RUNPATH (ohne DT_RPATH)[2]
4. /etc/ld.so.cache
5. /lib/
6. /usr/lib

Mac OS X

1. @execute_path
2. $DYLD_LIBRARY_PATH
3. @rpath[4]
4. /lib
5. /usr/lib
6. $DYLD_FALLBACK_LIBRARY_PATH
7. public Frameworks

Versionsinformationen

Windows	Linux	Mac OS X
in der Library einkodiert	libNAME.so.MAJOR.MINOR.BUILD	libNAME.MAJOR.MINOR.BUILD.dylib

[1] Falls eine Library mit absoluter Pfadangabe gelinkt wird, erübrigt sich das Suchen zur Laufzeit
[2] Die Linker Option -Wl,-rpath erzeugt das ELF Symbol DT_RPATH, standardmässig wird damit auch das Symbol DT_RUNPATH erzeugt, was man mit –disable-new-dtags ausschalten kann
[3] bei s-uid und s-gid Programmen greift diese Umgebungsvariable nicht
[4] wobei rpath wiederum @execute_path oder @loader_path beinhalten kann

Libraries without hell III

Bei C++ muss die Abwärtskompatibilität innerhalb einer Major-Release auf kleinstem gemeinsamen Compiler-Nenner gewahrt bleiben.

Keine Änderung an den Zugriffsrechten oder Inhalten einer Memberfunktion

Keine public/protected Funktion nach inline überführen

Keine public/protected Funktionen entfernen, die nicht inline sind

Keine privaten Datenmember hinzufügen/entfernen (-> m_pData)

Keine Veränderungen/Entferen von public/protected Members

Keine Änderung an der Klassenhierachie (nur neue Kinder)

Kein Hinzufügen/Entfernen von virtuellen Memberfunktionen (vtbl der Subklasse)

Möglichst kein Hinzufügen von Reimplementationen von virtuellen Member

(SuperClass::virtualFunction wird beim Kompilieren und nicht zur Laufzeit evaluiert)

Auch die OPTIONAL Statements in Fortran, bzw. Ellipsenfunktionen in C können funktionell leicht die Kompatibilität brechen, obgleich zumindest der Programmstart gewährleistet ist.

Libraries without hell IV

Mischen von Fortran und C/C++

Unter beinahe allen UNIX Compilern werden die Fortan Entrypoints in Kleinschreibung mit abschließendem _ generiert,deshalb

```
#ifdef UNIX
#define MYFUNCTION myfunction_
#endif
```

Die unter C++ aufzurufende Methode wird als C Methode deklariert.
FORTRAN ist ein optionales Makro und steuert die Aufrufkonvention (cdcecl & Co.)
alle Werte müssen mit call-by-reference übergeben werden

```
extern "C" void FORTRAN MYFUNCTION(int *i,int* j,....)
```

Ab Fortran 2003 ist es wesentlich einfacher C Funktionen einzubinden:

`bind( c )`
: bind-Attribut, stellt unter anderem die Interoperabilität mit C bezüglich Prozedurnamenskonventionen nach der Übersetzung sicher

`use, intrinsic :: iso_c_binding`
: Einbindung des intrinsischen Moduls iso_c_binding

`real( kind = c_float )`
: C-Datentyp float.

`value`
: call by value

Compiler Issues I

Größe einer Datenstruktur nicht zuverlässig determinierbar

```
struct foo{
    int i;
    char c[2];
    short s;
}
```

durch gerichtete Alignments kann folgende Regel verletzt(!) werden

```
sizeof(foo) = sizeof(foo.i) + sizeof(foo.c) + sizeof(foo.s)
```

enum muss nicht zwangsläufig ein *int* sein, die Wahrscheinlichkeit ist aber recht hoch, da heutzutage auf dem Stack übergebene Parameter auf 32bit ausgerichtet werden.

```
enum Color RED;
setColor(RED); // ACHTUNG falls ein int-Argument erwartet wird
```

Lösungen sind:

```
enum Color { RED=0x00000001 };
```

Compiler Schalter der *enum=int* erzwingt

Bedingte Anweisungen (if) erfordern je nach Compiler explizit einen logischen Datentyp (bool, .LOGICAL.) und akzeptieren keinen integer!

Compiler Issues II

Kommentar in Kommentar kann zu Fehler führen (mind. Warnung)

Bitshifts ist nicht Vorzeichensicher (Implementationsabhängig)

Mit Pragmas können spezielle Compilereinstellungen im Code eingestellt werden. Mit C99 gibt es die ersten Standard Pragmas

```
#pragma STDC [name] [option]

#pragma STDC FP_CONTRACT ON
```
(Fließkommaausdrücke zusammenfassen)

Nur auf C89 / C99 definierte Makros (Präprozessor defines) verlassen. C89 definiert lediglich 5 Makros, mit C99 sind es bereits 10. Hier eine Auswahl:

`__STDC__`	konforme Implementierung = 1
`__STDC_VERSION__`	genaue C Version (C99)
`__FILE__`	aktuelle Quelldatei
`__LINE__`	aktuelle Zeilennummer
`__func__`	vordefinierter Bezeichner (C99)

einfache(?!) Datentypen

TYP	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
int	32	64	32	32	16
long	64	64	32	32	32
long long		64			
pointer	64	64	64	32	32

TYP bezeichnet unterschiedliche Programmierkonventionen, die auf die Größe der C Kerndatentypen hinweisen.

heutige 32bit Umgebungen benutzen durchweg: ILP32
64bit Windows verwendet: LLP64 (P64)
64bit UNIX verwendet: LP64

64bit Ganzzahlen sind in C<99 uneinheitlich:
    Compiler aus der UNIX Welt verwenden: long long
    Compiler der Windows Welt verwenden: __int64

klare Datentypen

eigene defines verwenden

ANSI C99 Datentypen

TYP	Beschreibung
int8_t	vorzeichenbehafteter 8-Bit-Integer
uint8_t	vorzeichenloser 8-Bit-Integer
int16_t	vorzeichenbehafteter 16-Bit-Integer
uint16_t	vorzeichenloser 16-Bit-Integer
int32_t	vorzeichenbehafteter 32-Bit-Integer

Fortran

TYP	Beschreibung
INTEGER(KIND=2)	vorzeichenbehafteter 16-Bit
INTEGER(KIND=4)	vorzeichenbehafteter 32-Bit (default)
INTEGER(KIND=8)	vorzeichenbehafteter 64-Bit

Für Speicherbereiche **size_t** verwenden (normalerweise typedef auf unsigned int)

Prozessor Internas I

Manche Prozessorerweiterungen, bspw. Eine Vektoreinheit, verlangen explizit eine Ausrichtung (*Alignment*)
Ausrichtungen via union (C) bzw. EQUIVALENCE (Fortran)

```
union{
        float scalars[vec_step(vector float)];
        vector float v;
}buffer;
```

Beispiel der PowerPC AltiVec Einheit, **vector** ist Schlüsselwort auf PPC Plattform
scalars werden gesetzt, **v** wird an den Aufrufer geliefert..

Byte-Order bei binären Transfer von Daten

Big-Endian	höherwertiges Bit zuerst
Little-Endian	niederwertigstes Bit zuerst

-> Bibliotheken verwenden (**QDataStream**)

TCP/IP überträgt via Big-Endian
-> keine direkte Weitergabe in die Netzwerkschicht
-> High-Level Zugriff verwenden (**QNetwork** Klassen)

Prozessor Internas II

Byte Order: Codebeispiel:

```
union{
    long l; // long = 4Byte
    unsigned char c[4];
}u;
u.l = 0x12345678;
printf("c[0]=0x%x\n",(unsigned)u.c[0]);
```

Adresse	Little-Endian	Big-Endian
&c[0]	0x78	0x12
&c[1]	0x56	0x34
&c[2]	0x34	0x56
&c[3]	0x12	0x78

Implementation des sicheren Schreibens:

```
void write_ulong(FILE *fp,unsigned long u){
    c[0] = (unsigned char) (u >> 24 );
    c[1] = (unsigned char) (u >> 16 );
    c[1] = (unsigned char) (u >> 8 );
    c[1] = (unsigned char) (u );
    fwrite(c,sizeof(c),1,fp);
}
```

Implementation Lesen analog
effizienter mit Byte-Order Erkennungsmethode (analog union-bsp. oben)

Prozessor Internas: IEEE 754

Standardisierter Umgang mit Fließkommazahlen

einfache Genauigkeit: 32bit

Bit:	0-23	Mantisse/Nachkommateil (implizit führende 0)
Bit:	24-30	Exponent
Bit:	31	Vorzeichenbit (0=positiv,1=negativ)

doppelte Genauigkeit: 64bit

Bit:	0-51	Mantisse/Nachkommateil (implizit führende 0)
Bit:	52-62	Exponent
Bit:	63	Vorzeichenbit (0=positiv,1=negativ)

Erweiterte Genauigkeit erreicht 80bit. Aktuelle FPU's (*Floating Point Units*) rechnen mit 128bit, haben aber spezielle Instruktionen für SIMD

Seit C99 und Fortran 2003 standardisierter Zugriff auf die FPU

Fortran 2003: **ieee_exceptions**,**ieee_arithmetic**,**ieee_features** (intrinsic)
C99: **fenv.h** (header)

Zugriff auf 5 Ausnahmeergebnisse (Überlauf,Unterlauf,dbz,Unzulässig,Ungenau), Darstellung umschalten, Rundungsart einstellen

Zusätzliche Optimierungsmöglichkeiten, bei C casts von Gleitkommazahlen zu int:

Standardmässig wird die FPU gezwungen den Zustand mit einer Flush-Operation abzuspeichern (**_ftol**) was das Memory kurzzeitig blockiert.
Abschalten dieses Verhaltens mit Compiler Switches (Intel: /Qifist) oder Assembler Code (Portabilitätsprobleme).

Tools & Tipps

Zeilenendenproblematik den Revisionssystemen (**subversion,git,..**) überlassen

OS/FS Spezifikas auch über Revisionssysteme (**subversion,git,..**) machbar
    (links,svn:executable)

Dateinamen besser Konservativ (ASCII Zeichen als kl. gemeinsamer Nenner)
    Fileserver *können* Konvertierung
    USB-Stick,scp *können es nicht*

Quoting bei Skripten nicht vergessen
    falsch: `cp $SOURCE $TARGET`
    richtig: `cp "$SOURCE" "$TARGET"`

verschiedene Sicherheitsmodelle berücksichtigen (POSIX,ACL Rechte...)

Zusammenfassung

Vieles blieb unerwähnt (z.B. QListBox Annahmen, wchar_t)

i18n ist so ein großes Thema, dass es getrennt betrachtet werden sollte

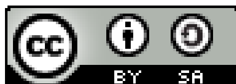
Man kann sich auf nicht allzuviel verlassen

Lauffähig auf einem oder gar zwei Systemen heißt nicht automatisch portabel

verbindliche Standards stellen die Basis dar, auf der programmiert werden kann

portable code homepage :

http://www.hookatooka.com/wpc/



some rights reserved by Harald Nikolisin