35

C-Header

Dieses Kapitel behandelt die folgenden Themen:

- Auswahl der nach C++ übernommenen C-Header
- Unterschiede und Alternativen

Dieser Abschnitt zeigt die von der Programmiersprache C übernommenen Header, soweit sie nicht obsolet sind. Der Inhalt ist derselbe wie in der C-Standard Library [ISOC]. Die Dateinamen ergeben sich aus den Header-Namen nach den üblichen Konventionen. Die Datei zum Header `<cmath>` heißt dementsprechend *math.h*. Viele der C-Header sind gut und mit Beispielen in [Her] beschrieben. In diesem Abschnitt *nicht* ausführlich behandelt werden:

- `<cfloat>`: Wegen des Headers `<limits>` (siehe Seite 725) ist die Verwendung von `<cfloat>` nicht mehr notwendig.
- `<ciso646>`: Der Header `<ciso646>` existiert nur aus Gründen der Kompatibilität zu C. *iso646.h* definiert Makros für nationale Tastaturen, die bestimmte Sonderzeichen nicht zur Verfügung haben. Zum Beispiel ist `and` ein Makro für `&&`. Die C-Makros sind Schlüsselwörter in C++ (siehe Tabelle auf Seite 887), weswegen `<ciso646>` im Allgemeinen nicht mehr gebraucht wird – und die Makros auch nicht mehr enthält.

- `<climits>`: Wegen des Headers `<limits>` (siehe Seite 725) ist die Verwendung von `<climits>`, der Grenzwerte für ganzzahlige Datentypen festlegt, im Allgemeinen nicht mehr notwendig.
- `<clocale>`: Wegen des Headers `<locale>` (siehe Seite 821) ist die Verwendung von `<clocale>` im Allgemeinen nicht mehr notwendig.
- `<csetjmp>`: Mit den Funktionen `setjmp()` und `longjmp()` des Headers `<csetjmp>` lassen sich Sprünge über Funktionsgrenzen realisieren, mit denen Fehlersituationen in den Funktionen abgefangen werden. Die Funktionen sind dank des in Kapitel 8 beschriebenen Exception-Handlings von C++ im Allgemeinen nicht mehr notwendig.
- `<csignal>`: Im Header `<csignal>` sind Signale und die zugehörigen Funktionen zur Behandlung deklariert. Signale sind Unterbrechungen (englisch *interrupts*), die durch Soft- oder Hardware erzeugt werden, wenn besondere Ausnahmesituationen auftreten, wie zum Beispiel Division durch Null. Signale werden zum Großteil wegen des in Kapitel 8 beschriebenen Exception-Handlings von C++ im Allgemeinen nicht mehr benötigt. Eine ausführliche Beschreibung ist in [Her] zu finden.

35.1 <cassert>

Zusicherungen (englisch *assertions*) werden mit dem Header `<cassert>` eingebunden. Einzelheiten siehe Abschnitt 3.3.5 auf Seite 133.

35.2 <cctype>

Der Header `<cctype>` enthält die in den Tabellen 35.1 und 35.2 aufgeführten C-Funktionen zum Klassifizieren und Umwandeln von Zeichen. Maßgeblich ist die aktuell gültige `locale`-Einstellung, siehe dazu Kapitel 31.

Tabelle 35.1: Umwandlungsfunktionen aus `<cctype>`

Schnittstelle	Bedeutung
`tolower(z)`	gibt `z` als Kleinbuchstaben zurück
`toupper(z)`	gibt `z` als Großbuchstaben zurück

Weil in C sowohl Zeichen als auch Wahrheitswerte vom Datentyp `int` sind, sind die Parameter und Rückgabewerte der Funktionen in Tabelle 35.1 vom Typ `int`. Der C++-Compiler nimmt die Umwandlung nach `bool` bzw. von oder nach `char` automatisch vor. Nicht darstellbare Zeichen der Tabelle 35.2 sind in Hexadezimalnotation geschrieben (vergleiche mit der ASCII-Tabelle auf Seite 887 f.). Die beiden Funktionen der Tabelle 35.2 geben ihr Argument unverändert zurück, wenn sich eine Umwandlung erübrigt. Unter dem Hea-

der <cwctype> finden sich die entsprechenden Funktionen für »wide character«-Zeichen. isalnum() heißt dort iswalnum(), und Entsprechendes gilt für die anderen Funktionen.

Tabelle 35.2: Klassifizierungsfunktionen aus <cctype>

Schnittstelle	wahr, wenn z ==	Bereich
isalnum(z)	Buchstabe oder Ziffer	A..Z, a..z, 0..9
isalpha(z)	Buchstabe	A..Z, a..z
isblank(z)	Leerzeichen (C locale)	
iscntrl(z)	Steuerzeichen	0x00..0x1f, 0x7f
isdigit(z)	Ziffer	0..9
isgraph(z)	druckbares Zeichen (ohne ' ')	0x21..0x7e
islower(z)	Kleinbuchstabe	a..z
isprint(z)	druckbares Zeichen (mit ' ')	0x20..0x7e
ispunct(z)	druckbar, aber weder ' ' noch alphanumerisch	0x21..0x2f, 0x3a..0x40 und 0x5b..0x7e
isspace(z)	Zwischenraumzeichen	0x09..0x0d
isupper(z)	Großbuchstabe	A..Z
isxdigit(z)	hexadezimale Ziffer	0..9, A..F, a..f

35.3 <cerrno>

Im Header <cerrno> wird eine globale Variable errno deklariert, deren Wert von vielen Systemfunktionen im Fehlerfall gesetzt wird. Der zugehörige Fehlertext wird als char* von der Funktion strerror(int) (Header <cstring>) zurückgegeben. Beispiel: cout << strerror(errno);. Der Verwendung von errno ist die in Kapitel 8 beschriebene Ausnahmebehandlung vorzuziehen.

35.4 <cmath>

Die mathematischen Funktionen der folgenden Tabelle 35.3 sind im Header <cmath> für Grunddatentypen zu finden. Es gibt einige Ausnahmen: Die Funktionen abs() für Ganzzahlen, div(), rand() und srand() sind aus historischen Gründen unter dem Header <cstdlib> (siehe unten) abgelegt. Hinweis: Manche Funktionen, die laut Text eine Ganzzahl zurückliefern sollen, liefern den Wert dieser Zahl als *Gleitkommazahl* zurück (Beispiel: ceil()).

Tabelle 35.3: Mathematische Funktionen

Schnittstelle	Mathematische Entsprechung
`F abs(F x)`	$\lvert x \rvert$
`F acos(F x)`	$\arccos x$
`F asin(F x)`	$\arcsin x$
`F atan(F x)`	$\arctan x$
`F atan2(F x, F y)`	$\arctan(x/y)$
`F ceil(F x)`	kleinste Ganzzahl größer oder gleich x
`F cos(F x)`	$\cos x$
`F cosh(F x)`	$\cosh x$
`F exp(F x)`	e^x
`F fabs(F x)`	$\lvert x \rvert$
`F floor(F x)`	größte Ganzzahl kleiner oder gleich x
`F fmod(F x, F y)`	Rest der Division x/y
`F frexp(F x, int* pn)`	zerlegt eine Zahl x in die Mantisse m und den Exponent n. `pn` ist ein Zeiger auf den Exponenten n, d.h. n = `*pn`. Es gilt $0.5 \leq m < 1$ und $x = m \cdot 2^n$.
`F ldexp(F x, int n)`	$x \cdot 2^n$
`F log(F x)`	$\ln x$
`F log10(F x)`	$\log_{10} x$
`F modf(F x, F* i)`	zerlegt x in einen ganzzahligen Anteil und den Rest. Dabei ist `*i` der ganzzahlige Anteil von x. Der restliche Bruchteil wird zurückgegeben.
`F pow(F x, F y)`	x^y
`F pow(F x, int y)`	x^y
`F sin(F x)`	$\sin x$
`F sinh(F x)`	$\sinh x$
`F sqrt(F x)`	$+\sqrt[2]{x}$
`F tan(F x)`	$\tan x$
`F tanh(F x)`	$\tanh x$

Abkürzung: `F` = einer der Typen `float`, `double` oder `long double`

35.5 <cstdarg>

Funktionen mit Argumentlisten variabler Länge enthalten eine Ellipse (drei Punkte als Auslassungszeichen) in der Parameterliste, etwa `(int, ...)`. Die Funktionen benötigen die Datentypen und Makros des Headers `<cstdarg>`. Ein Beispiel für eine Ellipse in der Parameterliste ist die C-Funktion `printf()`, ein anderes ist auf Seite 306 zu finden. Von Funktionen dieser Art wird grundsätzlich abgeraten, weil die Typprüfung der Aufrufparameter durch den Compiler außer Kraft gesetzt ist.

35.6 <cstddef>

Der Header <cstddef> enthält Standarddefinitionen des jeweiligen Systems (Tabelle 35.4).

Tabelle 35.4: Standarddefinitionen aus <cstddef>

Name	Bedeutung
size_t	vorzeichenloser Ganzzahltyp für das Ergebnis von sizeof
ptrdiff_t	ganzzahliger Typ mit Vorzeichen zur Subtraktion von Zeigern
wchar_t	Typ für die auf Seite 51 erwähnten »wide characters«
offsetof	Abstand eines Strukturelements vom Strukturanfang in Bytes
NULL	Null-Zeiger (dasselbe wie 0 oder 0L in C++)

35.7 <cstdio>

Die im Header <cstdio> abgelegten Ein- und Ausgabefunktionen sind wegen der iostream-Bibliothek im Allgemeinen nicht mehr notwendig. Deswegen werden sie hier nicht aufgeführt. Es gibt aber Ausnahmen, weil C++ keine Bibliothek für die Dateiverwaltung (löschen, umbenennen, Verzeichnis lesen und anlegen usw.) hat. Diese Ausnahmen werden jedoch von der Boost-Library abgedeckt. Abschnitt 25.1 gibt einen Überblick über die wichtigsten Funktionen, teilweise auf cstdio basierend, teilweise auf Boost.

35.8 <cstdlib>

Zunächst seien in Tabelle 35.5 die in Tabelle 35.3 fehlenden mathematischen Funktionen aufgeführt, die aus historischen Gründen zum Header <cstdlib> gehören. div_t ist eine vordefinierte Struktur, die das Divisionsergebnis und den Rest enthält. Die long-Variante dieser Struktur ist ldiv_t.

```
struct div_t { int quot; int rem; }; // Quotient, Rest (remainder)
```

Die wichtigsten anderen Funktionen zeigt Tabelle 35.6. Die Speicherverwaltungsfunktionen sind weggelassen worden, weil sie wegen new und delete nicht mehr notwendig sind.

Tabelle 35.5: Mathematische Funktionen aus `<cstdlib>`

Schnittstelle	Mathematische Entsprechung
`int abs(int x)`	$\|x\|$
`long abs(long x)`	$\|x\|$
`long labs(long x)`	$\|x\|$
`div_t div(int z, int n)`	Struktur `div_t` (siehe Text)
`ldiv_t div(long z, long n)`	Struktur `ldiv_t` (siehe Text)
`ldiv_t ldiv(long z, long n)`	Struktur `ldiv_t` (siehe Text)
`int rand()`	Pseudozufallszahl zwischen 0 und `RAND_MAX`. `RAND_MAX` ist die größtmögliche Pseudozufallszahl.
`void srand(unsigned seed)`	initialisiert den Zufallszahlengenerator

Tabelle 35.6: Ausgewählte Funktionen aus `<cstdlib>`

Schnittstelle	Bedeutung
`void abort()`	Programmabbruch
`void atexit(void (*f)())`	trägt die Funktion `f` in eine Liste von Funktionen ein, die vor dem normalen Programmende aufgerufen werden.
`void exit(int status)`	normales Programmende. Der Status wird an das Betriebssystem gemeldet.
`int system(const char* B)`	Befehl `B` an den Kommandointerpreter des Betriebssystems geben
`char* getenv(const char* E)`	Wert der Environmentvariablen `E`
`int atoi(const char* s)`	interpretiert den C-String `s` als `int`-Zahl
`long int atol(const char*s)`	dito als `long`-Zahl
`double atof(const char* s)`	dito als `double`-Zahl
`long strtol(const char* s, char** rest, int basis)`	interpretiert den C-String `s` als `long`-Zahl. `basis` gibt die Basis an. Ein ggf. nicht konvertierbarer Rest wird in `rest` abgelegt. Ausführliche Beschreibung und Anwendungsbeispiel siehe Seite 627.
`unsigned long strtoul(const char* s, char** rest, int basis)`	dito für `unsigned long`
`double strtod(const char* s, char** rest)`	dito für `double`
`void* bsearch(const void* key, const void* base, size_t n, size_t size, int (*cmp)(const void*, const void*))`	binäre Suche. `key` zeigt auf den Schlüssel, der im Feld `base` mit `n` Elementen gesucht wird. Die Größe der Feldelemente ist `size`, die Vergleichsfunktion ist `cmp`.
`void qsort(void*, size_t, size_t, int (*)(const void*, const void*))`	Quicksort (siehe Seite 224)

35.9 <cstring>

Obwohl die C++-String-Klasse existiert, sind die auf `char*` basierenden C-Strings unerlässlich, weswegen die wichtigsten Funktionen des Headers `<cstring>` hier beschrieben werden. Außer Funktionen für C-Strings sind Funktionen für Bytefelder enthalten, die nicht mit '\0' abgeschlossen sein müssen.

Abweichung vom C-Standard

In Abweichung vom C-Standard treten manche Prototypen doppelt auf, siehe zum Beispiel `strchr()`, aber auch viele andere. Der Grund liegt in der gewünschten besseren Typsicherheit. So möchte man evtl. mit `strchr()` die Adresse eines Zeichens in einer konstanten Zeichenkette ermitteln. Ein Rückgabetyp `char*` statt `const char*` würde die `const`-Eigenschaft beseitigen, und der Compiler würde sich beschweren. Andererseits möchte man evtl. mit `strchr()` die Adresse eines Zeichens in einer Zeichenkette ermitteln, die anschließend modifiziert werden soll. Dann wäre der Typ `const char*` ungeeignet. Das Vorhandensein beider Varianten gestattet es dem Compiler, die zum Typ des Parameters passende auszusuchen.

Funktionen für C-Strings (Auswahl)

- `char* strcat(char* s1, const char* s2)` kopiert die Zeichenkette `s2` an das Ende von `s1` und liefert `s1` zurück. Das Nullbyte am Ende von `s1` wird überschrieben. Die Zeichenketten dürfen sich nicht überlappen.
- `const char* strchr(const char* s, int c)` und
  `char* strchr( char* s, int c)`
  geben die Adresse des Zeichens `c` zurück, falls es im Feld `s` einschließlich '\0' gefunden wurde, ansonsten ist das Ergebnis NULL.
- `int strcmp(const char* s1, const char* s2)`
  vergleicht die Speicherbereiche `s1` und `s2`. Es wird 0 zurückgegeben, wenn kein Unterschied festgestellt wird. Falls das erste unterschiedliche Zeichen von `s1` kleiner als das entsprechende in `s2` ist, wird eine negative Zahl zurückgegeben; falls es größer ist, eine positive Zahl.
- `char* strcpy(char* z, const char* q)`
  kopiert alle Bytes bis einschließlich des Nullbytes von der Quelle `q` zum Ziel `z`. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich nicht überlappen.
- `int strcspn(const char* s1, const char* s2)`
  gibt die größte Länge einer Zeichenkette vom Anfang bis zu der Position in `s1` an, bis zu der kein in der Zeichenkette `s2` vorhandenes Zeichen gefunden wird.
- `char* strerror(int N)`
  liefert den Text der Systemfehlermeldung mit der Nummer `N` als C-String (vergleiche `errno` auf Seite 875).
- `size_t strlen(const char* s)`
  gibt die Anzahl der Zeichen (ohne Nullbyte) im C-String `s` zurück.

- `const char *strpbrk(const char* s1, const char* s2)` und
  `char *strpbrk( char* s1, const char* s2)`
  liefern die Position in `s1`, an der erstmalig ein Zeichen aus `s2` gefunden wird. Falls nichts gefunden wird, ist das Funktionsergebnis NULL.
- `const char* strrchr(const char* s, int c)` und
  `char* strrchr( char* s, int c)`
  liefern die Position in `s`, an der letztmalig das Zeichen `c` gefunden wird. Falls nichts gefunden wird, ist das Funktionsergebnis NULL.
- `const char *strstr(const char* s1, const char* s2)` und
  `char *strstr( char* s1, const char* s2)`
  liefern die Position in `s1`, an der erstmalig die Zeichenkette `s2` (ohne '`\0`') gefunden wird. Falls nichts gefunden wird, ist das Funktionsergebnis NULL.
- `char* strtok(char* s, const char* trenn)`
  Die Zeichenkette `s` wird in sogenannte Tokens zerlegt, wobei die möglichen Trennzeichen, durch die die Tokens getrennt sind, durch `trenn` definiert sind. Die Funktion gibt in aufeinanderfolgenden Aufrufen (mit dem Argument NULL ab dem 2. Aufruf) jeweils die Adresse des nächsten Tokens an, sofern eins gefunden wird. Der C-String `s` wird dabei modifiziert, indem die Trennzeichen mit '`\0`' überschrieben werden. Ein Beispiel ist auf Seite 643 zu sehen.

Funktionen für C-Strings maximaler Länge

- `char* strncat(char* s1, const char* s2, size_t n)`
  kopiert die Zeichenkette `s2` an das Ende von `s1` und liefert `s1` zurück. Das Nullbyte am Ende von `s1` wird überschrieben. Es werden maximal `n` Zeichen geschrieben. Die Zeichenketten dürfen sich nicht überlappen.
- `int strncmp(const char* s1, const char* s2, size_t n)`
  vergleicht maximal `n` Zeichen der Speicherbereiche `s1` und `s2`. Es wird 0 zurückgegeben, wenn kein Unterschied festgestellt wird. Falls das erste unterschiedliche Zeichen von `s1` kleiner als das entsprechende in `s2` ist, wird eine negative Zahl zurückgegeben; falls es größer ist, eine positive Zahl.
- `char* strncpy(char* z, const char* q, size_t n)`
  kopiert die Bytes bis einschließlich des Nullbytes, aber maximal `n`, von der Quelle `q` zum Ziel `z`. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich nicht überlappen.

Funktionen für Bytefelder

- `const void* memchr(const void* s, int c, size_t n)` und
  `void* memchr( void* s, int c, size_t n)`
  geben die Adresse des Bytes `c` zurück, falls es in den ersten `n` Positionen des Feldes `s` gefunden wurde, ansonsten ist das Ergebnis NULL.
- `int memcmp(const void* s1, const void* s2, size_t n)`
  vergleicht maximal `n` Positionen der Speicherbereiche `s1` und `s2`. Es wird 0 zurückgegeben, wenn kein Unterschied festgestellt wird. Falls das erste unterschiedliche Byte

von s1 kleiner als das entsprechende in s2 ist, wird eine negative Zahl zurückgegeben; falls es größer ist, eine positive Zahl.

- `void* memcpy(void* z, const void* q, size_t n)`
  kopiert n Bytes von der Quelle q zum Ziel z. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich nicht überlappen.
- `void* memmove(void* z, const void* q, size_t n)`
  kopiert n Bytes von der Quelle q zum Ziel z. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich überlappen.
- `void* memset(void* s, int c, size_t n)`
  initialisiert die ersten n Positionen des Feldes s mit dem Byte c. Die Adresse s wird zurückgegeben.

35.10 <ctime>

Der Header <ctime> enthält verschiedene Funktionen zur Bearbeitung und Auswertung der Systemzeitinformation (Anwendungsbeispiel siehe Seite 335).

Datentypen

Neben den bekannten Typen NULL und size_t gibt es die Datentypen clock_t (für CPU-Zeiten), time_t (für Datums- und Zeitangaben) und struct tm, eine Struktur mit mindestens den Elementen der Tabelle 35.7:

Tabelle 35.7: Struktur tm

tm-Element	Bedeutung
int tm_sec	Sekunden 0 .. 59
int tm_min	Minuten 0 .. 59
int tm_hour	Stunden 0 .. 23
int tm_mday	Monatstag 1 .. 31
int tm_mon	Monat 0 ..11
int tm_year	Jahr seit 1900
int tm_wday	Wochentag seit Sonntag 0 .. 6
int tm_yday	Tag seit 1. Januar 0 .. 365
int tm_isdst	*is d*aylight saving *t*ime, Werte: Sommerzeit (> 0), Winterzeit (0), undefiniert (-1)

Funktionen

- `char* asctime(const tm*)`
  `char* ctime(const time_t*)`
  Die Funktionen wandeln die im tm-Format oder time_t-Format vorliegende Zeit in einen formatierten C-String um. Der C-String endet mit \n\0, beendet also die laufende Zeile.

- `clock_t clock()`
  gibt die seit Programmstart verstrichene CPU-Zeit in »Ticks« zurück. Die Ticks können in Sekunden umgerechnet werden, wenn der von `clock()` zurückgegebene Wert durch die vordefinierte Konstante `CLOCKS_PER_SEC` dividiert wird.
- `double difftime(time_t t1, time_t t2)`
  ermittelt die Differenz beider Zeiten in Sekunden.
- `size_t strftime(char* buf, size_t max, const char* format, const tm* z)`
  Die Funktion wandelt die im `tm`-Format vorliegende Zeit in einen formatierten C-String um, wobei das Ergebnis im Puffer `buf` abgelegt wird und `format` einen C-String mit Formatvorgaben darstellt. Die Anzahl der in den Puffer geschriebenen Zeichen wird zurückgegeben, falls sie < `max` ist, ansonsten ist das Ergebnis der Funktion 0. Die im Unixsystem möglichen Formate können in der Shell mit `man strftime` erfragt werden. Beispiele für übliche Formate: "%j" gibt den Tag des Jahres aus, "%c" Datum und Uhrzeit, "%x" nur das Datum. Die Formate können im Formatstring aneinandergehängt werden. Nicht als Format interpretierte Zeichen werden direkt übertragen.
- `tm* gmtime(const time_t* z)`
  `tm* localtime(const time_t* z)`
  Beide Funktionen wandeln die in `*z` vorliegende Zeit in die Struktur `tm` um. Dabei gibt `localtime()` die lokale Ortszeit unter Berücksichtigung von Sommer- und Winterzeit zurück, während `gmtime()` die UTC (Universal Time Coordinated, entspricht GMT (Greenwich Mean Time)) zurückgibt. Die UTC in einem Unixsystem basiert auf der Zahl der seit dem 1.1.1970 verstrichenen Sekunden.
- `time_t mktime(const tm*)`
  wandelt eine Zeit im `tm`-Format in das `time_t`-Format um.
- `time_t time(time_t* z)`
  gibt die momentane Kalenderzeit zurück bzw. –1 bei Fehler. Falls `z` $\neq$ NULL ist, wird der Rückgabewert an der Stelle `z` hinterlegt.