

Bemerkungen zu C++:

In C++ kann man fünf Teilsprachen, die bezüglich Syntax und Semantik differieren, unterscheiden.

Es sind:

C-Sprache

Objektorientierte Erweiterungen von C

Templates

Standardbibliothek

Präprozessorsprache

Bemerkung: Das C++-Komitee versucht, die Fähigkeiten der Präprozessorsprache von C weitgehend durch C++-Konstrukte zu ersetzen.

Erstes Beispiel:

```
// Ausgeben und Einlesen von Zeichenketten
// Vorbereitungen unter Windows:
// 1. Schriftart der Eingabeaufforderung
// ändern, z. B. in Lucida Console
// 2. chcp 1252 ausführen
```

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main () {
    cout << "Ausgabe einiger Umlautzeichen: \n"
           "ÄÖÜß - äöüß - áêû - " << endl;
    std::string st; // Der Vorsatz std:: stört nicht
    std::cout << "st: ";
    // Einlesen einer Zeile
    getline (cin, st);
    cout << "st = " << st << endl;
} //main
```

/* Beispiellauf:

```
Ausgabe einiger Umlautzeichen:
ÄÖÜß - äöüß - áêû -
st: áúý - àòè - ÄÖÜß
st = áúý - àòè - ÄÖÜß
```

```
*/
```

Zeichen:

Aus dem Standard:

The fundamental storage unit in the C++ memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

Für das Schreiben von C++-Programmen gibt es den "basic source character set", er entspricht in etwa den druckbaren Zeichen des 7-Bit-ASCII-Codes und einigen Kontrollzeichen. Neben diesem Grundzeichensatz existieren: "basic execution character set", "basic execution wide-character set", "execution character set", "execution wide-character set". Sie sind alle implementations-definiert.

Die 91 graphischen Zeichen des "basic source character set":

```
abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
0123456789  
_{}[]#()<>%:; .? * + - / ^ & | ~ ! = , \ " ' "
```

Man kennt drei Zeichenarten, char, signed char, unsigned char. Die Mitglieder dieser drei Arten sind typverschieden. Daneben gibt es noch den Typ wchar_t.

Beispiele für Zeichen:

```
'a', '\141', '\x61', 'A', '\101', '\x41'
```

Sonderformen zur Zeichendarstellung:

newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
formfeed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"

Daneben existieren für die Angabe eines Zeichens die Oktaldarstellung, die Sedezimaldarstellung und zwei universelle Darstellungen, \ooo, \xhhhh, \uhhhh und \Uhhhhhhhh.

Einige Zeichen der Codepage Windows 1252:

```
cout << "'a', '\141', '\x61', 'A', '\101', '\x41 \n'"  
"'x80', '\xe0', '\xe1', '\xe2', '\xe3', '\xf0', '\xf1 \n'";
```

Ausgabe: 'a', 'a', 'a', 'A', 'A', 'A'
'€', 'à', 'á', 'â', 'ã', 'ð', 'ñ'

Bezeichner:

identifizier:

identifizier-nondigit
identifizier identifizier-nondigit
identifizier digit

identifizier-nondigit:

nondigit
universal-character-name
other implementation defined character

nondigit: one of

a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _

digit: one of

0 1 2 3 4 5 6 7 8 9

Bemerkungen:

- (i) Ein Bezeichner beginnt mit einem Unterstrich oder einem Buchstaben, gefolgt von Buchstaben, Ziffern oder Unterstrich.
- (ii) Welche Unicode-Zeichen als Buchstaben gelten, ist in dem Dokument TR 10176:2003 geregelt.
- (iii) Manche Bezeichner sind für die Sprache C++ und ihre Bibliotheken reserviert. Dazu gehören die Schlüsselwörter und Bezeichner, die zwei aufeinanderfolgende Unterstriche enthalten.
- (iv) Bezeichner unterliegen keiner Längenbeschränkung.

```
// Umlaute in Bezeichnern
// Beispiel unter Windows
// Benutzung in Visual C++
#include <iostream>
using namespace std;
```

```
int main () {
    int gröÙe = 10;
    int grün = 20;
    cout << "grün + gröÙe = " << grün + gröÙe << endl;
}
/* Ausgabe: grün + gröÙe = 30 */
```

```
// Obiges Programm mit
// Universal Character Names
// Übersetzt mit g++ 4.3.3
// unter -fextended-identifiers
#include <iostream>
using namespace std;
```

```
int main () {
    int gr\u00F6\u00DFe = 10;
    int gr\u00FCn = 20;
    cout << "gr\u00FCn + gr\u00F6\u00DFe = "
        << gr\u00FCn + gr\u00F6\u00DFe << endl;
}
/* Ausgabe: gr |n + gr |fe = 30 unter cp 850
Transformation aus UTF-8 Codierung nach Unicode:
grün + gröÙe = 30
*/
```

Logischer Datentyp:

Werte: true, false

Operationen: !, &&, ||, ==, !=, =

Bemerkung: Es finden automatische Typanpassungen zwischen Booleschen Werten und Ganzzahlen statt.

// Beispiel zum Booleschen Datentyp

```
#include <iostream>
```

```
using namespace std;
```

```
int main () {
```

```
    bool a = true;
```

```
    bool b = false;
```

```
    bool c = 3 != 7;
```

```
    cout << "Ausgabe von Wahrheitswerten (Textform): "
```

```
        << endl
```

```
        << boolalpha
```

```
        << "    a = " << a << endl
```

```
        << "    b = " << b << endl
```

```
        << "    c = " << c << endl << endl;
```

```
    cout << "Ausdruck: (10 < 5 == 23 > 77) = "
```

```
        << (10 < 5 == 23 > 77) << endl << endl;
```

```
} //main
```

```
/* Ausgabe:
```

```
Ausgabe von Wahrheitswerten (Textform):
```

```
    a = true
```

```
    b = false
```

```
    c = true
```

```
Ausdruck: (10 < 5 == 23 > 77) = true    */
```

Ganzzahlen:

C++ kennt zwei Arten von Ganzzahlen: vorzeichenbehaftete und vorzeichenlose Ganzzahlen.

vorzeichenbehaftete Ganzzahlen:

signed char,

short int,

int,

long int,

long long int.

vorzeichenlose Ganzzahlen:

unsigned char,

unsigned short int,

unsigned int,

unsigned long int,

unsigned long long int.

Bemerkungen:

- (i) Die Zahlbereiche für char, short, int, long und long long werden durch die Implementation festgelegt.
- (ii) Die vorzeichenbehafteten und vorzeichenlosen Varianten benötigen jeweils den gleichen Speicherplatz.
- (iii) Für die Zahlbereiche gilt die folgende Inklusionskette: signed char \subseteq short \subseteq int \subseteq long \subseteq long long.
- (iv) Die vorzeichenlosen Zahlen kennen nicht das Problem des Überlaufs.
- (v) Für Ganzzahlen sind die üblichen Operationen definiert.

```

// Zahlbereiche für vorzeichenbehaftete Zahlen
// außer wchar_t
#include <iostream>
#include <limits>          // Bereichsangaben
using namespace std;

int main() {
cout << "signed char umfasst die Zahlen " << endl
    << "von "
    << (int) numeric_limits<signed char>::min()
    << endl << "bis "
    << (int) numeric_limits<signed char>::max()
    << endl << endl;

cout << "short int umfasst die Zahlen " << endl << "von "
    << numeric_limits<short>::min() << endl << "bis "
    << numeric_limits<short>::max()
    << endl << endl;

cout << "int umfasst die Zahlen " << endl << "von "
    << numeric_limits<int>::min() << endl << "bis "
    << numeric_limits<int>::max()
    << endl << endl;

cout << "long int umfasst die Zahlen " << endl
    << "von "
    << numeric_limits<signed long int>::min() << endl
    << "bis " << numeric_limits<long>::max()
    << endl << endl;

cout << "long long int umfasst die Zahlen " << endl
    << "von " << numeric_limits<long long>::min()
    << endl << "bis "
    << numeric_limits<long long>::max()
    << endl << endl;

```

```

cout << "wchar_t umfasst die Zahlen " << endl
    << "von " << numeric_limits<wchar_t>::min()
    << endl << "bis "
    << numeric_limits<wchar_t>::max()
    << endl << endl;
} //main

```

/* Ausgabe: Ergebnisse für Visual C++

```

signed char umfasst die Zahlen
von -128
bis 127

```

```

short int umfasst die Zahlen
von -32768
bis 32767

```

```

int umfasst die Zahlen
von -2147483648
bis 2147483647

```

```

long int umfasst die Zahlen
von -2147483648
bis 2147483647

```

```

long long int umfasst die Zahlen
von -9223372036854775808
bis 9223372036854775807

```

```

wchar_t umfasst die Zahlen
von 0
bis 65535

```

*/

```

// Zahlbereiche für vorzeichenlose Zahlen
#include <iostream>
#include <string>
#include <limits>          // Bereichsangaben
using namespace std;

// Nutzung einer Schablone
template <typename tn>
void zahlbereich (string s) {
    cout << s + " umfasst die Zahlen " << endl << "von "
         << numeric_limits<tn>::min() << " bis "
         << numeric_limits<tn>::max()
         << endl << endl;
} // zahlbereich

// Sonderfall
void zahlbereich () {
    cout << "unsigned char umfasst die Zahlen " << endl
         << "von " << (int) numeric_limits<unsigned
                    char>::min() << " bis "
         << (int) numeric_limits<unsigned char>::max()
         << endl << endl;
} // zahlbereich

int main() {
    zahlbereich ();
    zahlbereich <unsigned short> ("unsigned short");
    zahlbereich <unsigned> ("unsigned");
    zahlbereich <unsigned long> ("unsigned long");
    zahlbereich <unsigned long long> ("unsigned long long");
    zahlbereich <wchar_t> ("wchar_t");
} // main

```

```

/* Ausgabe: Ergebnisse für Visual C++

```

```

unsigned char umfasst die Zahlen
von 0 bis 255

```

```

unsigned short umfasst die Zahlen
von 0 bis 65535

```

```

unsigned umfasst die Zahlen
von 0 bis 4294967295

```

```

unsigned long umfasst die Zahlen
von 0 bis 4294967295

```

```

unsigned long long umfasst die Zahlen
von 0 bis 18446744073709551615

```

```

wchar_t umfasst die Zahlen
von 0 bis 65535

```

```

*/

```

```

// Schreibweise von Ganzzahlen
// in Programmtexten
// Suffixe: u, U, l, L, ll, LL, ul, UL, ull, ULL
#include <iostream>
using namespace std;

int main () {

    int a = 123;
    int b = 0123;
    int c = 0x123;
    long la = 123456789l;
    long lb = 1234567892L;

    long long lla = 12345678901234ll;
    long long llb = 1234567890123456LL;

    unsigned ua = 0xfabcdefau;
    unsigned ub = -1U;
    unsigned uc = 4294967395LL;    // Warnung ?

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << "la = " << la << endl;
    cout << "lb = " << lb << endl;
    cout << "lla = " << lla << endl;
    cout << "llb = " << llb << endl;

    cout << "ua = " << ua << endl;
    cout << "ub = " << ub << endl;
    cout << "uc = " << uc << endl;
} //main

```

```

/* Ausgabe:

a   = 123
b   = 83
c   = 291
la  = 123456789
lb  = 1234567892
lla = 12345678901234
llb = 1234567890123456
ua  = 4206681850
ub  = 4294967295
uc  = 99                // Man beachte die
                        // Verkürzung von uc !!

*/

```

```

// Implizite Typanpassungen, Integral promotion
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {

signed char sca = 102, scb = 112, scc = 122;
cout << "sca = " << (int) sca << endl;
cout << "scb = " << (int) scb << endl;
cout << "scc = " << (int) scc << endl;

cout << "sca + scb + scc = " << sca + scb + scc << endl;
cout << "sca * scb * scc = " << sca * scb * scc << endl;
cout << "sca * scb * scc * sca * scb * scc = "
    << sca * scb * scc * sca * scb * scc << endl;
cout << "sca * scb * scc * sca * scb * scc * scc = "
    << sca * scb * scc * sca * scb * scc * scc << endl;

unsigned char uca = 102, ucb = 112, ucc = 122;

cout << "uca + ucb + ucc = " << uca + ucb + ucc << endl;
cout << "uca * ucb * ucc = " << uca * ucb * ucc << endl;
cout << "uca * ucb * ucc * uca * ucb * ucc = "
    << uca * ucb * ucc * uca * ucb * ucc << endl;
cout << "uca * ucb * ucc * uca * ucb * ucc * ucc = "
    << uca * ucb * ucc * uca * ucb * ucc * ucc << endl;

int ia = 2000111222, ib = 2111000333;

cout << "ia = " << ia << endl;
cout << "ib = " << ib << endl;
cout << "ia * ib = " << ia * ib << endl;

```

```

long long lla = ia;
cout << "lla * ib = " << lla * ib << endl;

unsigned ua = 2;
cout << "ua = " << ua << endl;
cout << "ua * -2 = " << ua * -2 << endl;

cout << typeid (ua*lla).name () << endl;

} //main

/* Ausgabe:

sca = 102
scb = 112
scc = 122
sca + scb + scc = 336
sca * scb * scc = 1393728
sca * scb * scc * sca * scb * scc = 1152520192
sca * scb * scc * sca * scb * scc * scc = -1126457344
uca + ucb + ucc = 336
uca * ucb * ucc = 1393728
uca * ucb * ucc * uca * ucb * ucc = 1152520192
uca * ucb * ucc * uca * ucb * ucc * ucc = -1126457344
ia = 2000111222
ib = 2111000333
ia * ib = -842072578
lla * ib = 4222235455679036926
ua = 2
ua * -2 = 4294967292
__int64

*/

```

Bitoperatoren:

Man kennt die folgenden Bitoperatoren:

~	Bitweise Negation
<<	Linksschieben
>>	Rechtsschieben
&	Bitweises UND
^	Bitweises XOR
	Bitweises ODER
<<=	Zuweisendes Linksschieben
>>=	Zuweisendes Rechtsschieben
&=	Zuweisendes bitweises UND
^=	Zuweisendes bitweises XOR
=	Zuweisendes bitweises ODER

```
#include <iostream>
int main() {
    std::cout << "Groß- und Kleinbuchstaben in der "
              << "ASCII-Codierung haben einen Abstand von 32."
              << std::endl;

    for (int i=65; i<91; ++i) {
        char ch = i;
        char chX32 = ch^32;
        std::cout << i << ":\\t" << ch << "\\t\\t" << int(chX32)
                  << ":\\t" << chX32 << "\\t\\t" << int(chX32^32)
                  << ":\\t" << char(chX32^32) << std::endl;
    }
} //main
```

/*

Groß- und Kleinbuchstaben in der ASCII-Codierung
haben einen Abstand von 32.

65:	A	97:	a	65:	A
66:	B	98:	b	66:	B
67:	C	99:	c	67:	C
68:	D	100:	d	68:	D
69:	E	101:	e	69:	E
70:	F	102:	f	70:	F
71:	G	103:	g	71:	G
72:	H	104:	h	72:	H
73:	I	105:	i	73:	I
74:	J	106:	j	74:	J
75:	K	107:	k	75:	K
76:	L	108:	l	76:	L
77:	M	109:	m	77:	M
78:	N	110:	n	78:	N
79:	O	111:	o	79:	O
80:	P	112:	p	80:	P
81:	Q	113:	q	81:	Q
82:	R	114:	r	82:	R
83:	S	115:	s	83:	S
84:	T	116:	t	84:	T
85:	U	117:	u	85:	U
86:	V	118:	v	86:	V
87:	W	119:	w	87:	W
88:	X	120:	x	88:	X
89:	Y	121:	y	89:	Y
90:	Z	122:	z	90:	Z

*/

Gleitpunktzahlen:

C++ kennt drei Genauigkeitsstufen für Gleitpunktzahlen: float, double und long double. Die Genauigkeiten der einzelnen Gleitpunkttypen sind implementationsabhängig.

Bemerkung: Für maschinelle Gleitpunktzahlen gelten die algebraischen Gesetze der Arithmetik nur eingeschränkt. Im Header <limits> findet man einige implementationsabhängige Konstanten, um eine sinnvolle Fehlerabschätzung zu initiieren.

```
#include <iostream>
#include <limits>
using namespace std;

int main() {
    float a = 1.234567E-7F,
          b = 1.000000,
          c = -b;
    float s1 = a + b;
    s1 += c;
    float s2 = a;
    s2 += b + c;
    cout << "Beispiel zur Nichtassoziativität der "
          "Gleitpunktoperationen" << endl << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
```

```
cout << "c = " << c << endl;
cout << "Summe (a+b)+c = " << s1 << endl; // 1.192e-7
cout << "Summe a+(b+c) = " << s2 << endl << endl;
// 1.234e-7
```

```
// Bestimmung der Zahlengenauigkeit.
// Die Zahlengenauigkeit ist implementationsabhängig.
cout << "Genauigkeit von float:      "
      << std::numeric_limits<float>::digits10 <<
      " Dezimalziffern" << endl;
cout << "Genauigkeit von double:    "
      << std::numeric_limits<double>::digits10 <<
      " Dezimalziffern" << endl;
cout << "Genauigkeit von long double: " <<
      std::numeric_limits<long double>::digits10 <<
      " Dezimalziffern" << endl;
```

```
}//main
```

```
/* Ausgabe: Rechnung auf Sparc unter Solaris
Beispiel zur Nichtassoziativität der Gleitpunktoperationen
a = 1.23457e-07
b = 1
c = -1
Summe (a+b)+c = 1.19209e-07
Summe a+(b+c) = 1.23457e-07

Genauigkeit von float:      6 Dezimalziffern
Genauigkeit von double:    15 Dezimalziffern
Genauigkeit von long double: 33 Dezimalziffern

*/
```

In C99 besteht die Möglichkeit, Gleitpunktkonstanten hardwarenah anzugeben. Das folgende Beispiel wurde mit dem Gnu C-Compiler auf Sun-Rechnern ausgeführt. Die Ausgabefunktion printf wurde um die Formate %a und %A erweitert.

```
#include <stdio.h>

int main (void) {

    double da = 0x1.3p+3;
    double db = 0x1.369p+3;
    double dc = 0x1.369a5p+3;
    double dd = 0x1.369a59ep+3;

    printf("da = %a \tda = %10.15f\n", da, da);
    printf("db = %a \tdb = %10.15f\n", db, db);
    printf("dc = %a \tdc = %10.15f\n", dc, dc);
    printf("dd = %a \tdd = %10.15f\n", dd, dd);

} //main

/* Ausgabe

da = 0x1.3000000000000p+3    da = 9.500000000000000
db = 0x1.3690000000000p+3    db = 9.705078125000000
dc = 0x1.369a500000000p+3    dc = 9.706336975097656
dd = 0x1.369a59e000000p+3    dd = 9.706341683864594

*/
```

Zeiger:

Zeiger zeigen auf Objekte oder Funktionen.

```
// Nutzung von Zeigern
#include <iostream>
using namespace std;
int main () {
    int wert = 1024;
    int* pi = &wert; // pi zeigt auf Ganzzahl
    int** ppi = &pi; // ppi zeigt auf einen Zeiger, der auf
                    // eine Ganzzahl zeigt
    cout << "Zugriff auf Wert:" << endl
         << "direkt:      " << wert << endl
         << "indirekt:     " << *pi << endl
         << "zweifach indirekt: " << **ppi << endl;
    int i = 4;
    int j = 10;
    int* p1 = &i;
    int* p2 = &j;
    *p2 = *p1 * *p2;
    *p1 *= *p1;
    cout << "j = " << j << endl;
    cout << "i = " << i << endl;
} //main

/* Ausgabe:
Zugriff auf Wert:
direkt:          1024
indirekt:        1024
zweifach indirekt: 1024
j = 40
i = 16
*/
```

```
// Zeiger und const
// Beispiel aus Standard
```

```
const int ci = 10;
const int* pc = &ci;
const int* const cpc = pc;
const int **ppc;
int i, *p, *const cp = &i;
```

```
int main () {
    i = ci;
    *cp = ci;
    pc++;
    pc = cpc;
    pc = p;
    ppc = &pc;
```

```
// Examples of ill-formed operations are
```

```
// ci = 1;      // error
// ci++;       // error
// *pc = 2;    // error
// cp = &ci;   // error
// cpc++;     // error
// p = pc;    // error
// ppc = &p;   // error:
//             // invalid conversion from 'int**'
//             // to 'const int**'
```

```
*ppc = &ci; // OK, but would make p point to ci ...
// ... because of previous error
```

```
*p = 5; // clobber ci
```

```
//main
```

```
// Zur Zeigerarithmetik
// Zeiger +/- Ganzzahl, Zeiger - Zeiger
#include <iostream>
using namespace std;
```

```
int main () {
    const int arr_sz = 5;
    int arr [arr_sz] = { 0, 1, 2, 3, 4 };
    // pbegin zeigt auf erstes Element,
    // pend auf Element unmittelbar hinter dem letzten
    for (int* pbegin = arr, *pend = arr + arr_sz;
         pbegin != pend; ++pbegin)
        cout << *pbegin << ' ';
    cout << endl;
    cout << "Nochmalige Ausgabe des Arrays arr:"
         << endl;
    int* barr = &arr [0];
    cout << *(arr + 0) << ' '
         << *(1 + arr) << ' '
         << 2 [arr] << ' '
         << *(barr + 3) << ' '
         << *(4 + barr) << endl;
    cout << endl;
}
```

```
/* Ausgabe:
```

```
0 1 2 3 4
```

```
Nochmalige Ausgabe des Arrays arr:
```

```
0 1 2 3 4
```

```
*/
```

```

// Arrays

/* Beispiele eindimensionaler Arrays */

#include <iostream>
#include <iomanip>
#include <typeinfo>
using namespace std;

int main () {
    // Definition eines neuen Arrays
    int arr01 [] = {-21, -43, -65, 76, 87, 91};
    cout << "Ausgabe des Arrays arr01:"
         << endl;
    for (int i = 0; i < sizeof arr01 / sizeof arr01 [0]; ++i) {
        cout << " Element " << i << ":"
             << setw (6) << arr01 [i] << endl;
    }
    cout << endl;
    cout << "Typ von arr01 = " << typeid (arr01).name()
         << endl;
    cout << "Ausgabe von arr01: " << arr01 << endl;
    cout << "Typ von &arr01 [0] = " <<
         typeid (&arr01[0]).name() << endl;

    // Dynamische Erzeugung eines Arrays
    int* ap01 = new int [6];
    // Initialisierung
    for (int i = 0; i < 6; ++i)
        ap01 [i] = 10*i + i;
    // Ausgabe
    cout << endl

```

```

         << "Erste Ausgabe des ap01-Arrays:"
         << endl;
    for (int i = 0; i < 6; ++i)
        cout << setw (6) << ap01 [i];
    cout << endl;
    cout << "Zweite Ausgabe des ap01-Arrays:"
         << endl;
    for (int i = 0; i < 6; ++i)
        cout << setw (6) << *(ap01 + i);
    cout << endl;
    // Speicher freigeben
    delete [] ap01;
} //main

```

/* Ausgabe:

Ausgabe des Arrays arr01:

```

Element 0: -21
Element 1: -43
Element 2: -65
Element 3:  76
Element 4:  87
Element 5:  91

```

Typ von arr01 = int [6]

Ausgabe von arr01: 006BFDE0

Typ von &arr01 [0] = int *

Erste Ausgabe des ap01-Arrays:

```

0 11 22 33 44 55

```

Zweite Ausgabe des ap01-Arrays:

```

0 11 22 33 44 55

```

*/

```

/* Array als Parameter*/

#include <iostream>
#include <iomanip>
#include <typeinfo>
using namespace std;

int arr01 [6] = {-21, -43, -65, 76, 87, 91};
int arr02 [3] = {32, 42};

void f (int a []) {
    cout << "Typ von a = " << typeid(a).name() << endl;
}

int main () {
    // Aufruf von f
    f (arr01);
    f (arr02);
    // Sonderbehandlung von char [ ]
    char a [ ] = "Erste Zeichenkette";
    char* b = "Zweite Zeichenkette";

    cout << a << endl;
    cout << b << endl;
} //main

/* Ausgabe:
Typ von a = int *
Typ von a = int *
Erste Zeichenkette
Zweite Zeichenkette
*/

```

```

/* Array: Typanpassung */

#include <iostream>
#include <typeinfo>
using namespace std;

int arr01 [6] = {-21, -43, -65, 76, 87, 91};
int arr02 [3] = {32, 42};

void f (int a []) {}
// void f (int *) {} Fehler: Neudefinition von f

typedef int * ta;
typedef int tb [];

int main () {
    cout << typeid (ta).name() << endl;
    cout << typeid (tb).name() << endl;
    if (typeid (ta) != typeid (tb))
        cout << typeid (ta).name() << " und " <<
            typeid (tb).name() << " sind verschieden." << endl;
    cout << arr01 << endl;
    cout << arr02 << endl;
} //main

/* Ausgabe:
int *
int [0]
int * und int [0] sind verschieden.
00425004
0042501C
*/

```

// Beispiel zu Referenzen

```
#include <iostream>
using namespace std;

int main () {
    int i;
    int& ri = i; // Referenzen müssen initialisiert werden!
    int& rri = ri;
    i = 5; ri = 10; rri = 25;
    cout << "i = " << i << ", ri = " << ri
        << ", rri = " << rri << endl;

    long long a [] = {1234, 5678, 9012, 3456, 7654};
    long long& ra0 = a [0];
    long long& ra3 = a [3];

    cout << "a [0] = " << ra0 << endl;
    cout << "a [3] + 10 = " << ra3 + 10 << endl;
}// main

/* Ausgabe:
i = 25, ri = 25, rri = 25
a [0] = 1234
a [3] + 10 = 3466

*/
```

Bemerkung: Eine Referenz stellt einen Alias für ein Objekt dar. Es existieren keine Referenzen auf Referenzen, keine Arrays von Referenzen und keine Zeiger auf Referenzen. Es ist un spezifiziert, ob eine Referenz Speicherplatz benötigt.

// Weiteres Beispiel zu Referenzen

```
#include <cstdlib>
#include <iostream>
using namespace std;

int a [20];

void f (int& x) {
    const int z = 10;
    x += z;
}

int& g (int i) {
    return a [i];
}

int main () {
    int i = 10;
    f (i); f (i);
    cout << "i = " << i << endl;
    g (4) = 40; g (7) = 77;
    cout << "Ausgabe von a:" << endl;
    for (int i = 0; i < 10; i++)
        cout << a [i] << " ";
    cout << endl;
}//main

/* Ausgabe:
i = 30
Ausgabe von a:
0 0 0 0 40 0 0 77 0 0
*/
```

```

#include <iostream>
void add (int&, int&, int&);
void add (int const&, int const&, int&);
//void add (int, int, int&);          // Mehrdeutigkeit

int main () {
    int erg;  add (1, 2, erg);
    std::cout << erg << '\n';
    int a = 2, b = 3;  add (a, b, erg);
    std::cout << erg << '\n';
    int c = 9;  int const d = 12;  add (c, d, erg);
    std::cout << erg << '\n';
} //main

void add (int& a1, int& a2, int& e) {
    std::cout << "add ohne const\n";  e = a1+a2;
} //add

void add (int const& a1, int const& a2, int& e) {
    std::cout << "add mit const\n";  e = a1+a2;
} //add

void add (int a1, int a2, int& e) {
    std::cout << "add simple\n";  e = a1+a2;
} //add

/*
add mit const
3
add ohne const
5
add mit const
21
*/

```