Einfache Datentypen

Übersicht

- Namensbildung
- Variable und Konstanten
- Deklaration und Definition
- Ein- / Ausgabe
  - Formatierung
- Zuweisung (Speicherstelle / Wert)
- Bibliotheksfunktionen
- Typumwandlung

Namen

alles braucht einen Namen

- Vorschriften für Namensbildung
  - bestehen aus Buchstaben, Ziffern und Unterstrichen "_"
  - müssen mit Buchstabe oder Unterstrich "_" beginnen
  - Groß- und Kleinbuchstaben werden unterschieden
  - Anzahl signifikanter Zeichen implementierungsabhängig (i.a. >30)
  - teilweise zusätzliche Einschränkungen für externe Namen (Linker, Assembler, andere Sprachen, ...)
- dringende Empfehlungen
  - nicht mit Unterstrich "_" beginnen
  - natürlichsprachig, problemnah, aussagekräftig
  - inhaltsleere Namen (i, n) allenfalls für lokale Schleifenzähler etc.
  - Vorsicht mit Abkürzungen: Verständlichkeit darf nicht leiden

reserviert für Bibliotheken

englisch und deutsch nicht mischen !

Beispiele für Namen

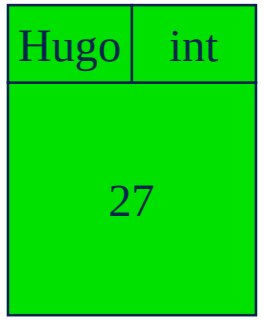
§ Merke: Namen sind **nicht** Schall und Rauch, sondern entscheidend für die Lesbarkeit von Programmen !

falsch	L	J
12xyz	s	Jahressumme
Zeile-1	cnt	counter
Zähler	_open	OpenFile
Datenfluß	ret	SetPaletteEntries
	atoi	ASCII_to_integer
	M	verschiebe_Fenster
		next_entry

bevorzugte Schreibweise bei mehreren Worten

Variablen und Konstanten

- Variablen sind Speicherzellen, die verschiedene Werte eines Typs *Objekte einer Klasse* speichern können
  - zu verschiedenen Zeitpunkten können Variablen verschiedene Werte enthalten
- Konstanten sind Speicherzellen, die während der gesamten Programmlaufzeit denselben Wert *dasselbe Objekt* enthalten
  - im Programmcode fest eingetragene Werte werden ebenfalls als Konstanten bezeichnet
- Beide sind Tripel aus Name, Typ und Speicherplatz
  - die Zuordnung erfolgt mittels einer Deklaration



Zahlentypen: Speicherbedarf und Wertebereiche

Klassen von Zahlen

ein **Datentyp** definiert, wie ein Bitmuster zu interpretieren ist

Klasse	Typ	Bits	signed *(default)* Min	signed *(default)* Max	unsigned Min	unsigned Max
ℤ	**char**	8	-128	127	0	255
	short int	16	-32768	32767	0	65535
	int (Win3.1)	16	-32768	32767	0	65535
	int (Win95)	32	-2147483648	2147483647	0	4294967295
	long int	32	-2147483648	2147483647	0	4294967295

Klasse	Typ	Bits	|Min|	|Max|	Genauigkeit e bei 1
ℝ	float	32	$3.4 * 10^{-38}$	$3.4 * 10^{38}$	$1.2 * 10^{-7}$
	double	64	$1.7 * 10^{-308}$	$1.7 * 10^{308}$	$2.2 * 10^{-16}$
	long double	80	$1.2 * 10^{-4932}$	$1.2 * 10^{4932}$	$1.1 * 10^{-19}$

(implementierungsabhängig; siehe limits.h und float.h; hier MS Visual C++)

Konstanten

Notation für konstante Objekte von Zahlenklassen

Anfang bestimmt Zahlenbasis
Endung bestimmt Datentyp

23103	**int**	**dezimal**
-23103L	long int	dezimal
23103Ul	unsigned long int	dezimal
030071	int	oktal
0x**5a3f**ul	unsigned long int	hexadezimal
0X**5A3F**	int	hexadezimal

12.34	**double**
-12.34f	float
12.34L	long double
1.3e-7	**double**

Codierung implementierungsabhängig
(DOS ASCII, Windows ANSI, ...)

'k'	**char**	**Zeichen**
'\23'	char	oktal
'\x3e'	char	hexadezimal
'\''	char	'
'\"'	char	"
'\a'	char	alert (Piep)
'\b'	char	backspace
'\f'	char	formfeed
'\n'	char	newline
'\r'	char	carriage return
'\t'	char	horizontal tab
'\v'	char	vertical tab

Deklaration, Definition und Initialisierung

so werden Objekte von Zahlenklassen erzeugt

Eine **Deklaration** macht die Variable dem Compiler nur bekannt;
eine **Definition** reserviert außerdem den Speicherplatz.

```
char Eingabe;                       // deklariert + definiert eine Variable

unsigned long int summe;            // ebenfalls nur eine Variable

short index, zustand;               // zwei Variablen vom selben Typ

double akkumulator = 0;             // definiert und initialisiert

const double pi = 3.14159,          // definiert 2 Konstanten
             e (2.71828);           // alternative Form der Initialisierung

const char newline ('\n');          // definiert eine Zeichenkonstante
```

optional

int

J

Ein- und Ausgabe

Austausch von Objekten von Zahlenklassen mit dem Benutzer

```
#include <iostream>                          // Standardbibliothek
using namespace std;
int    iPar = 27;                            // Deklaration
double dPar = 3.5;
cout << dPar << iPar << 3 << endl;           // ausgeben
cin  >> iPar >> dPar;                        // einlesen
cout << dPar << " " << iPar << endl;
```

Zeilenvorschub

Trennstelle

Ausgabe:	3.5273
Eingabe:	87 11.345
Ausgabe:	11.345 87

- *cout*: die Umwandlung in ein geeignetes Format erfolgt automatisch
  - abhängig vom Datentyp; der Operator << ist überladen
- *cin*: eingegebene Zeichenkette muß zum Typ der Argumente passen
  - sonst wird die Zeichenkette nicht weiter gelesen; (cin >> x) ist dann 0
  - Leerzeichen, Tab, Zeilenvorschub, Seitenvorschub und Return werden überlesen
  - Schreibweise der Zahlen: Zahlenbasis wie bei Konstanten, kein Suffix

Formatierung der Ausgabe von Zahlen

beeinflußt die Notation von Objekten von Zahlenklassen

- "Manipulatoren" für
  - Zahlenbasis
    `dec  hex  oct`
  - Anzahl der Schreibstellen (width)
    `setw (n)` // nur für folgende Zahl
    `setfill (c)` // vorlaufende Füllzeichen
  - Anzahl der signifikanten Stellen
    `setprecision (n)`
- werden in Ausgabestrom eingefügt
- bewirken keine Ausgabe, sondern eine Zustandsänderung des Ausgabestroms

```
#include <iomanip>
cout << hex << 127 << endl;
cout << dec << 127 << endl;
cout << oct << 127 << endl;

cout << setfill ('0')
     << setw(5) <<127<<endl;
cout << setw(2) <<127<<endl;

cout << setprecision (4)
     << 123.234567 << endl
     << 3.2 << endl;
cout << setw (10)
     << setprecision (4)
     << 4.5
     << endl;
```

```
7f
127
177
00177
177
123.2
3.2
00000004.5
```

keine Kennzeichnung der Basis

Formatierung von Gleitkommazahlen

```
#include <iomanip>
```

- Standard: Format abhängig vom Wert

```
cout << resetiosflags (ios::fixed)
     << resetiosflags (ios::scientific)
     << setprecision  (s);    // signifikante Stellen
```

- Festkommadarstellung -0.013471

```
cout << setiosflags   (ios::fixed)
     << resetiosflags (ios::scientific)
     << setprecision  (n);    // Nachkomma-Stellen
```

- Exponentialdarstellung -1.3471e-002

```
cout << resetiosflags (ios::fixed)
     << setiosflags   (ios::scientific)
     << setprecision  (n);    // Nachkomma-Stellen
```

Weitere <iomanip> Members

§ left / right	Ausgabe links/rechts falls Bereich noch nicht gefüllt
§ fixed	Ausgabe in fixed decimals ab diesem Zeitpunkt wirkt sich setprecision() auf die Nachkommastellen aus.
§ scientific	Ausgabe in wissenschaftlicher Schreibweise
§ uppercase nouppercase	Ausgabe der Buchstaben bei hex Ausgabe

Zuweisungen, Speicherzellen und Werte

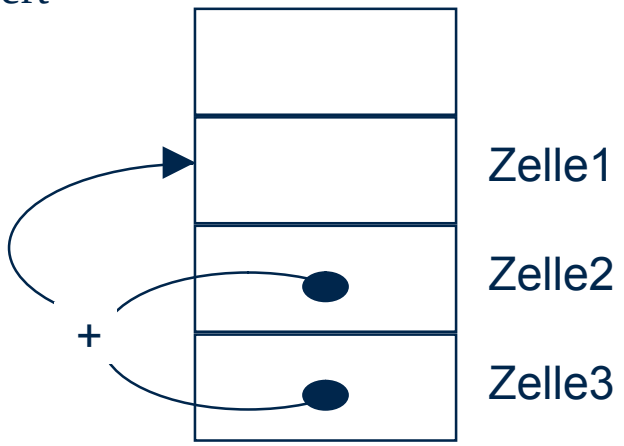
= Speicherzelle

- eine **Zuweisung** ändert den Inhalt einer Variablen
- ein **Ausdruck** liefert einen Wert; *man unterscheidet:*
  - **L-Werte** bezeichnen Speicherzellen
    - dürfen auf der linken Seite einer Zuweisung stehen
    - Variablennamen
    - Ausdrücke, die eine Speicheradresse liefern
  - **R-Werte** bezeichnen Konstanten und Speicherinhalte
    - rechte Seite einer Zuweisung
    - Variablennamen werden automatisch "dereferenziert"
    - Ausdruck, der eine Zahl o.ä. liefert

variable Objekte

konstante Objekte

```
int Zelle1, Zelle2, Zelle3;
Zelle1 = Zelle2 + Zelle3;   // ok
27 = Zelle2 + Zelle3;       // Unsinn
```



Typ bool für Wahrheitswerte

erst spät im Rahmen des ANSI/ISO-Standards hinzugekommen; in Visual C++ seit Version 5

- `bool` ist aus historischen Gründen ein Integertyp
  - automatische Typkonversion in beide Richtungen
  - 0 == false; 1 == true
- Vergleichsoperatoren ==, !=, <, <=, >, >= liefern `bool`
- Logische Verknüpfungen && (und), || (oder), ! (nicht)
- Bedingungen if, while, etc. erwarten strenggenommen `int`
  - alles != 0 bedeutet true
  - wir setzen grundsätzlich `bool`-Ausdrücke ein !

```
bool zuGross;

if (a > Schranke)
   zuGross = true;
else
   zuGross = false;

zuGross = (a > Schranke);

if (zuGross)
   cout << Meldung;
```

L

J

```
typedef int BOOL;
#define FALSE 0
#define TRUE  1
```

veralteter Behelf aus windows.h

Operatoren, die R-Werte liefern

d.h. sie dürfen nur auf der rechten Seite einer Zuweisung stehen

arithmetisch:

Addition	+
Subtraktion	-
Ä auch unär	
Multiplikation	*
Division	/
Modulo (Rest)	%
Ä nur für int	

Vergleich:

größer	>
größer oder gleich	>=
kleiner	<
kleiner oder gleich	<=
gleich	==
ungleich	!=

logisch:

und	&&
oder	||
nicht (unär)	!

basierend auf:

0 = false

!=0 = true

Besonderheit: Shortcircuit-Auswertung

Funktionen der Klasse bool

Funktionen der Zahlenklassen

fallende Priorität bei der Auswertung

Klammern verbessern die Übersicht:

L a + b >= c - d && e * f

J ((a+b) >= (c-d)) && (e*f)

Auswertereihenfolge in Ausdrücken

- die Auswertereihenfolge innerhalb von Ausdrücken ist grundsätzlich undefiniert !
  - man darf sich nicht darauf verlassen, ob in

    x = (a+b) * (c+d)

    zuerst (a+b) oder (c+d) ausgewertet wird

- Ausnahme: shortcircuit(= Kurzschluß)-Auswertung bei *&&* (logisches Und) und || (logisches Oder)
  - der 2. Operand wird nicht ausgewertet, wenn
    - bei *&&* der 1. Operand `false` ist
    - bei || der 1. Operand `true` ist

    dann steht der Wert des gesamten Ausdrucks nämlich schon fest
  - Beispiel: `wenn (esRegnet && keinSchirm) dann nass;`

    hier wird "keinSchirm" nicht ausgewertet, wenn es nicht regnet.

Kombinierte Zuweisungsoperatoren

jede Zuweisung ist in C++ zugleich ein Ausdruck

		Kurzform für	Wert des Ausdrucks
Inkrement nach Abfrage	x++		x
Inkrement vor Abfrage	++x	x=x+1	x+1
Dekrement nach Abfrage	x--		x
Dekrement vor Abfrage	--x	x=x-1	x-1
Addition mit Zuweisung	x+=y	x=x+y	
Subtraktion mit Zuweisung	x-=y	x=x-y	
Multiplikation mit Zuweisung	x*=y	x=x*y	
Division mit Zuweisung	x/=y	x=x/y	
Modulo mit Zuweisung	x%=y	x=x%y	

wenn in eine andere Anweisung eingebettet, z.B.
y = x++ ;

nicht zu empfehlen

L

im Normalfall eine freistehende Anweisung

Bibliotheksfunktionen

- Eine der Stärken von C/C++ sind die Standardbibliotheken

```
#include <math.h>
double x,y,z;

z=sin(x)        Sinus
z=cos(x)        Cosinus
z=tan(x)        Tangens
z=sinh(x)       Sinus Hyperbolicus
z=exp(x)        Exponentialfunktion
z=log(x)        Logarithmus
z=pow(x,y)      x^y
z=sqrt(x)       Quadratwurzel
z=fabs(x)       Absolutwert
...             viele weitere
```

```
#include <ctype.h>
int c;                  unsigned char
int i;                  ==0 oder !=0

i=isalpha(c)            Buchstabe
i=isdigit(c)            Ziffer
i=islower(c)            Kleinbuchstabe
i=isupper(c)            Großbuchstabe
i=isspace(c)            Leerst.,Tab., ...
i=isxdigit(c)           Hexadez. Ziffer
i=tolower(c)  Groß- > Kleinb.
i=toupper(c)  Klein- > Großb.
...                     viele weitere
```

Typumwandlung

Umwandlung zwischen Zahlenklassen

- automatisch *(abhängig von der Umgebung)*
  - Umwandlung in Typ mit größerem Wertebereich ist unproblematisch
    - char ð short ð int ð long ð float ð double ð long double
    - `int_wert + double_wert` konv. int_wert und ruft double-Addition auf
  - Umwandlung in Typ mit kleinerem Wertebereich verliert Genauigkeit
    - long double ð double ð float ð long ð int ð short ð char
    - Runden oder Abschneiden ist implementierungsabhängig
  - signed ó unsigned kann falsches Ergebnis liefern
- explizit durch "cast"-Operator

```
Wurzel_aus_2 = sqrt ((double) 2);
Wurzel_aus_2 = sqrt (double (2));
```

entsprechend o.g. Regeln, mit gleichen Problemen

funktionale Notation nur für Typ-/Klassen-Namen

Aufzählungstypen: enum

- definiert einen neuen Typ und Konstanten vom Typ `int`

```
enum Farbe {rot, gruen, blau};          // beginnend bei 0
enum Farbe {rot=7,gruen,blau=3};        // explizit und wahlfrei
```

- Deklaration einer Variablen

```
enum Farbe {rot,gruen,blau} Farbtopf;   // zugleich
Farbe Farbtopf;                         // separat; Farbe vorher definiert
Farbtopf = gruen;                       // Wertzuweisung
```

- automatische Konvertierung nach `int`

```
int Wert = Farbtopf;                    // ok: Farbtopf wird konvertiert
```

- keine direkte Wertzuweisung einer Integer; cast erforderlich

```
Farbtopf = Farbe (6);                   // zulässig, aber fragwürdig
```

d.h. der Programmierer übernimmt ausdrücklich die Verantwortung

Definition von Konstanten im Vergleich

- enum
  - Gruppe zusammengehöriger Ganzzahlkonstanten
- const
  - "nicht änderbare Variable"
  - Adreßoperator & anwendbar
  - dem Debugger bekannt
- #define
  - Substitution beliebiger Zeichenketten
  - nicht typsicher
  - **veraltet; nicht verwenden !**
    (in Bibliotheken recht häufig)

```
enum Aussage {falsch, wahr};
Aussage esRegnet;
esRegnet = wahr;
```

```
const int falsch = 0,
          wahr  = 1;
int esRegnet, *pAussage;
esRegnet = wahr;
pAussage = &falsch;
```

```
#define falsch 0
#define wahr   1
int esRegnet;
esRegnet = wahr;
```