

# Programmieren / Algorithmen & Datenstrukturen

Grundlagen (i), Teil 2



Prof. Dr. Skroch

**Universitatea**  
**BABEȘ-BOLYAI**

# Grundlagen (i)

Inhalt.

- ▶ Hallo C++
- ▶ Objekte, Typen, Werte, und Steuerungsprimitive
- ▶ Berechnungen und Anweisungen
- ▶ Fehler
- ▶ Fallstudie: Taschenrechner
- ▶ Funktionen und Programmstruktur
- ▶ Klassen

# Objekte, Ein- und Ausgaben

Programme lesen, verändern und schreiben Daten, die sich im Computerspeicher befinden.

## ► Programme

- lesen Eingabedaten,
- verändern Daten,
- schreiben Ausgabedaten.

## ► Um Daten lesen und schreiben zu können werden sie an einem "Ort" im Computerspeicher abgelegt.

## ► Einen solchen "Ort" nennen wir *Objekt*.

- Ein Objekt ist ein *Speicherbereich* mit einem *Typ*, wobei der Typ angibt, von welcher Art die dort abgelegten Daten sind.
- Ein *benanntes* Objekt wird auch *Variable* genannt.
  - Über den Namen des Objekts kann auf den Inhalt des Objekts zugegriffen werden.

# Objekte, Ein- und Ausgaben

Beispiele: ganze Zahl, Zeichenkette.

## ► Ganze Zahlen (Integer-Werte)

- Werden in Variablen vom *Typ* `int` abgelegt (ein im C++ Compiler integrierter Typ).
- Beispiel, ein `int` Objekt, das den *Namen* `yob` hat und den *Wert* `1997` enthält:

```
      int
yob  [1997]
```

## ► Zeichenketten (String-Werte)

- Werden in Variablen vom *Typ* `std::string` abgelegt (ein Typ aus der C++ StdLib).
- Beispiel, ein `std::string` Objekt, das den *Namen* `nick` hat und den *Wert* `capitalQ` enthält:

```
      std::string
nick  [capitalQ]
```

# Ganze Zahl und Zeichenkette ein- und ausgeben

Das Nickname Programm soll einen Spitznamen und das Geburtsjahr einlesen und beide Werte wieder ausgeben.

```
01  /*
02      nickname.cpp
03      yymdd-OSk
04      Liest einen Nickname ("Spitznamen") und das Geburtsjahr ein,
05      gibt die beiden eingelesenen Werte wieder aus.
06  */
07
08  #include <iostream>          // E/A-Stroeme aus der StdLib
09  using std::cin;             // using-Deklaration fuer cin aus std
10  using std::cout;           // using-Deklaration fuer cout aus std
11  #include <string>           // Strings aus der StdLib
12  using std::string;         // using-Deklaration fuer string aus std
13
14  int main( )
15  {
16      string nick { "capitalQ" }; // nick ist eine Variable vom Typ std::string
17      int      yob  { 1997 };      // yob (year of birth) ist eine Variable von Typ int
18
19      cout << "Bitte Nickname eingeben (gefolgt von \'Enter\'): ";
20      cin  >> nick; // lies die Zeichen in nick ein
21
22      cout << "Bitte Geburtsjahr eingeben (gefolgt von \'Enter\'): ";
23      cin  >> yob;  // lies die Zeichen in yob ein
24
25      std::clog << "\n\n\t" << "nick ist: " << nick
26                << "\n\t" << "yob ist: " << yob << "\n\n";
27      return 0;
28  }
```

# Zur Erinnerung (Grundlagen Teil 1)

Definition der benötigten Variablen.

► `std::string nick{"capitalQ"};`      `int yob{1997};`



- Die beiden Anweisungen in den Zeilen 16 und 17 definieren Variablen, eine vom Typ `std::string` namens `nick` und eine vom Typ `int` namens `yob`.
- Die beiden Anweisungen sind
  - *Deklarationen*, da sie *neue Namen* in das Programm einführen,
  - *Definitionen*, da sie auch *Speicher* für diese Variablen reservieren,
  - *Initialisierungen*, da sie die neuen Speicherbereiche direkt mit Werten vom passenden Typ füllen: `"capitalQ"` bzw. `1997`.
- Die beiden Variablen sind innerhalb der `main()`-Funktion des Nickname Programms definiert und (nur) dort gültig.

# Ganze Zahl und Zeichenkette ein- und ausgeben

Strom-Eingabe mittels der C++ Standardbibliothek (StdLib).

► `cin >> nick;            cin >> yob;`

► Der Standard-Eingabestrom `cin` und der dazu gehörende **Stromeingabe-Operator `>>`** der StdLib werden in Zeilen 20 und 23 verwendet.

- Der StdLib Stromeingabe-Operator `>>`

- liest die Zeichen vom Eingabestrom (linker Operand) ein und speichert sie in dem rechten Operanden ab.



- Unterscheidet den *Typ* des rechten Operanden, z.B. `bool`, `char`, `int`, `double`, `std::string`.


► Das Drücken der Enter-Taste ist nötig, um die Eingabe zu beenden.

- Der Zeilenumbruch, der mit Betätigung der Enter-Taste entsteht, ist *nicht* Teil der Zeichenkette, die im Speicher abgelegt wird, denn `>>` ignoriert Whitespace.


# Ganze Zahl und Zeichenkette ein- und ausgeben

Strom-Ausgabe mittels der C++ Standardbibliothek (StdLib).

- ▶ `cout << nick << "... " << yob << '\n';`
- ▶ Nachdem die Eingaben in `nick` und `yob` gespeichert sind, können sie im Programm verwendet (also z.B. wieder ausgegeben) werden.
- ▶ Wie im Hallo C++ Programm kann zur Ausgabe ein Standard-Ausgabestrom (etwa `std::cout` oder, wie in Zeilen 25-26, der Kontrollausgabestrom `std::clog`) mit dem Stromausgabe-Operator `<<` verwendet werden.
- ▶ Weitere Anmerkungen zum Quellcode:
  - Zeilen 19 und 22 geben einen Text aus, der den Benutzer auffordert, etwas einzugeben, ein solcher Text wird üblicherweise als *Eingabeaufforderung* bezeichnet.
  - Die Anführungszeichen werden für sog. Zeichenketten-*Literale* verwendet, die für sich selbst stehen und keine Namen haben – ohne Anführungszeichen bezieht sich ein Name auf den Wert von etwas mit diesem Namen, wie z.B. eine Variable:
    - `std::clog << "nick" << '\t' << nick << '\n'; // Kontrollausgabe`



Unbenanntes Zeichenketten-Literal  
Wert: `nick`



Variable namens `nick`  
Wert: `capitalQ`



# Variablen

Der C++ Compiler stellt zumeist sicher, dass eine Variable nur gemäß ihrem deklarierten Typ verwendet wird.

- ▶ *Variablen* sind benannte Objekte mit einem spezifischen Typ.
  - "Orte", an denen Daten gespeichert sind, heißen *Objekte*.
  - Um auf Objekte zuzugreifen, haben sie *Namen* (etwa `yob` oder `nick`).
  - Der *Typ* (etwa `int` oder `std::string`) legt fest:
    - 1) Was in dem Objekt abgelegt werden kann,
      - z.B. in einer `int`-Variable der Wert `1997`  
oder in einer `std::string`-Variable der Wert `"capitalQ"`.
    - 2) Welche Operationen damit ausgeführt werden können,
      - z.B. können `int`-Variablen mit dem Operator `*` multipliziert werden oder `std::string`-Variablen mit dem Operator `==` verglichen werden.
- ▶ Variablen dürfen nicht Werte vom falschen Typ beinhalten.


```
int yob { "acht" };           // falsch, "acht" ist kein int
std::string nick { "8" };    // ok, aber Vorsicht, "8" ist nicht 8
std::string s { 64 };        // ok, aber Vorsicht, die 64 steht hier fuer "\x40"
                             // ( auf vielen Systemen das at-Symbol @ )
```

# Typen

C++ und die StdLib stellen bereits eine recht große Auswahl an Typen zur Verfügung.

- ▶ Die fünf wohl wichtigsten Typen sind:

▪ <b>bool</b>	done	{ <b>false</b> }	}; // Boolesche Werte (false, true)
▪ <b>int</b>	answer	{ <b>42</b> }	}; // ganze Zahlen
▪ <b>double</b>	avg_grade	{ <b>4.2</b> }	}; // Gleitkommazahlen
▪ <b>char</b>	decimal_point	{ <b>'.'</b> }	}; // (einzelne) Zeichen
▪ <b>std::string</b>	nick	{ <b>"Kant"</b> }	}; // Zeichenketten



- ▶ bool, int, double und char sind integrierte C++ Typen, std::string ist ein sog. *benutzerdefinierter Typ* aus der C++ Standardbibliothek.
- ▶ Integrierte Typen haben eigene Formen für Literale.
  - Informieren Sie sich darüber selbstständig in einer C++ Referenztabelle.
- ▶ Zusätzlich zu ihrer Definition wird hier jeder der Variablen durch die sog. **`{}`-Initialisierung** auch ein Anfangswert zugewiesen.
  - Die Zuweisung eines sinnvollen Anfangswerts nennt man Initialisierung.
  - **Grundsätzlich sind *alle* Objekte *vor* der ersten Verwendung zu initialisieren.**
    - Etwa wie oben mit der `{}`-Syntax und passenden Literalen.
    - Wir werden noch andere Formen der Initialisierung kennen lernen, in vielen Fällen ist die Verwendung der oben gezeigten `{}`-Initialisierung die beste Lösung.

# Der StdLib Stromeingabe-Operator >> und Typen

Das Verhalten des Operators >> ist abhängig vom Typ der Variablen, in die eingelesen wird.

## ► Untersuchen Sie den Operator >> mittels des Nickname Programms...

- Das Lesen eines Eingabestroms vom linken Operand `std::cin` endet beim ersten Zeichen, das nicht mehr zum Typ des rechten Operanden (der Variablen) passt.
- Dieses Zeichen bleibt im Eingabestrom und ist das erste Zeichen, das für den nächsten rechten Operanden von diesem Eingabestrom gelesen wird (man kann sich vorstellen, dass es im `cin` Strom "wartet").
- Ist das erste Zeichen, das der Operator >> liest, für den jeweiligen Typ unzulässig,
  - dann geht der Eingabestrom `std::cin` in einen Fehlerzustand,
  - und dann bleiben alle folgenden Eingaben wirkungslos (d.h. die Werte der Variablen, in die als rechte Operanden des Operators >> eigentlich eingelesen werden sollte, bleiben unverändert).
- Whitespace:
  - Führender Whitespace im Eingabestrom wird vom Operator >> bei allen Typen ignoriert.
  - Beim Typ `std::string` beendet Whitespace im Eingabestrom per Konvention das Einlesen.

## ► Da sich der Operator >> für unterschiedliche Typen, auf die eingelesen wird (d.h. für unterschiedliche Typen seines rechten Operanden), jeweils spezifisch verhält, wird er auch als *überladener Operator* bezeichnet.

- In C++ sind viele Operatoren überladen (auch z.B. der Ausgabeoperator << ).

# Operationen und Typen

Der Typ einer Variablen legt fest, welche Operationen auf ihr ausführbar sind und wie diese Operationen angewandt werden.

## ► Beispiel `std::string` Typ und `int` Typ mit Operator `-` und Operator `+`

```
std::string first  { "Lukas" };
std::string second { "Cranach" };
int counter { 0 };

int c { counter + 1 }; // Operator + addiert int-Werte.
std::string name { first + ' ' + second }; // Operator + verknuepft string-Werte
// und kann auch char Werte verarbeiten.

// Der gleich unten verwendete Operator = (Zuweisungsoperator, mehr Details folgen)
// schreibt den Wert seines rechten Operanden in den Speicherplatz seines linken Operanden.
name = name + ", der Juengere"; // Operator + verknuepft string-Werte.
c = c - 1; // Operator - subtrahiert int-Werte.

//std::string s { name - ", der Juengere" }; // Fehler, der Operator - ist
// fuer std::string nicht definiert.
```

## ► Es handelt sich um reine *Syntax*-Vereinbarungen.

```
double age { 12.3333-25.0833 }; // Kein Syntaxfehler, obwohl es kein negatives Alter gibt.
std::string n { "Leibniz" + "Leibniz" }; // Kein Syntaxfehler, obwohl die
// Zeichenkette sinnlos aussieht.
```

# Zuweisung und Initialisierung

Der **Zuweisungsoperator** `=` wird verwendet, um einer Variablen (L-Wert) einen neuen Inhalt (R-Wert) zuzuweisen.

## ► Zuweisung mit Operator `=` beim `int`-Typ:

```
01 int main () {  
02     int a { 2 };    // Initialisierung: a mit Anfangswert 2  
03     int b { a };    // Initialisierung: b mit Anfangswert 2, aus a ermittelt  
04     a = 4;          // Zuweisung: a erhaelt den Wert 4  
05     b = 6 + a;      // Zuweisung: b erhaelt den Wert 10  
06     a = a + 8;      // Zuweisung: a erhaelt den Wert 12  
07     //c = 5;        // Fehler, c ist unbekannt  
08     //...  
09 }
```

## ► Der Operator `=` bedeutet *nicht* "ist gleich" sondern

- Zuweisung an *L-Werte* (linker Operand der Zuweisung), die ein Objekt bezeichnen,
- Zuweisung von *R-Werten* (rechter Operand der Zuweisung), die den Inhalt eines Objekts bezeichnen.

## ► Was geschieht bei der Zuweisung in Zeile 6 genau?

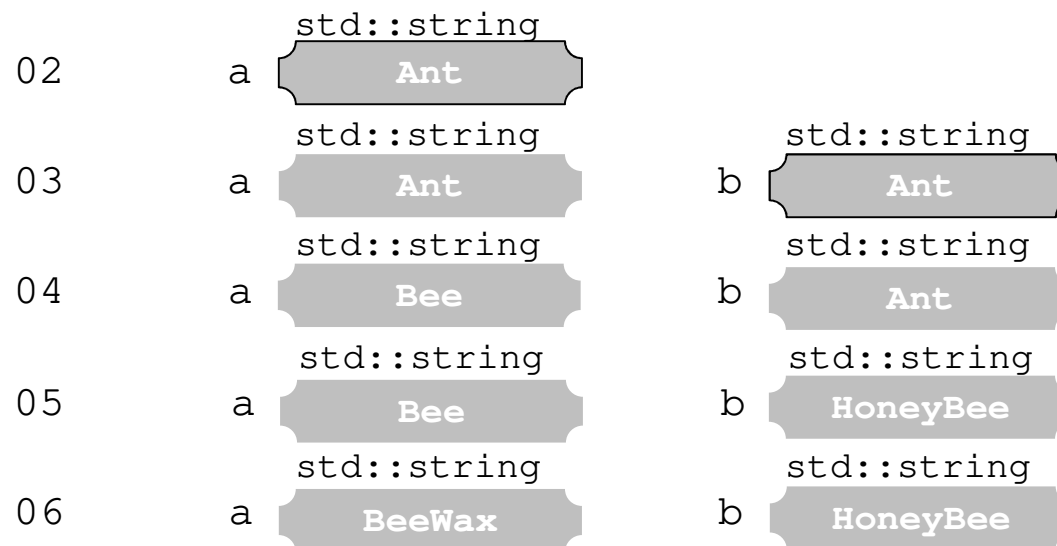
- 1) Der **R-Wert (d.h. Inhalt)** von `a` (rechts vom Operator `=`) wird ermittelt, Ergebnis ist 4.
- 2) Zu der 4 wird der Wert 8 des `int`-Literal addiert, Ergebnis ist 12.
- 3) Der Wert 12 wird nun in den **L-Wert (d.h. Objekt / Speicherplatz)** namens `a` (links vom Operator `=`) geschrieben.

# Zuweisung und Initialisierung

Der Zuweisungsoperator = anschaulich.

- Der Operator = funktioniert auch beim `std::string`-Typ.

```
01 int main () {  
02     std::string a { "Ant" }; // initialisiert: a mit Wert "Ant"  
03     std::string b { a };     // initialisiert: b mit Wert "Ant" (aus a)  
04     a = "Bee";               // zugewiesen: a (lval) erhaelt den Wert "Bee" (rval)  
05     b = "Honey" + a;         // zugewiesen: b (lval) erhaelt den Wert "HoneyBee" (rval)  
06     a = a + "Wax";           // zugewiesen: a (lval) erhaelt den Wert "BeeWax" (rval)  
07     //c = "Bug";             // Fehler: c ist unbekannt (kein L-Wert)  
08     //...  
09 }
```



# Wichtige Operatoren

Eine Übersicht über einige wichtige Operatoren für die fünf wohl am häufigsten benutzten Typen.

	bool	char	int	double	string
Zuweisung	=	=	=	=	=
Addition			+	+	
Verkettung (Konkatenation)					+
Subtraktion			-	-	
Multiplikation			*	*	
Division			/	/	
Modulo (Rest)			%		
Inkrement um 1			++	++	
Dekrement um 1			--	--	
Inkrement um n			+=n	+=n	
Anhängen					+=
Dekrement um n			--n	--n	
Multiplizieren und zuweisen			*=	*=	
Dividieren und zuweisen			/=	/=	
Modulo und zuweisen			%=		
Von s in x einlesen	s>>x	s>>x	s>>x	s>>x	s>>x
Von x nach s schreiben	s<<x	s<<x	s<<x	s<<x	s<<x
Gleich	==	==	==	==	==
Nicht gleich	!=	!=	!=	!=	!=
Größer als	>	>	>	>	>
Größer als oder gleich	>=	>=	>=	>=	>=
Kleiner als	<	<	<	<	<
Kleiner als oder gleich	<=	<=	<=	<=	<=

Bildquelle: Stroustrup.

*Vergleichsoperationen,*  
die alle mit den fünf  
wohl häufigsten Typen  
funktionieren.

# Vergleichsoperatoren und die `if`-Anweisung

Das Ergebnis von Vergleichsoperationen in Programmen dient meist dazu, den weiteren Ablauf des Programms zu steuern.

## ► Vergleich und `if`-Anweisung:

```
01 int main() {
02     std::string nick1 { "Jakob" };
03     std::string nick2 { "Esau" };
04
05     std::cout << "Geben Sie zwei Namen ein: ";
06     std::cin  >> nick1 >> nick2;
07
08     if( nick1 == nick2 )
09         std::cout << "Die Namen sind gleich\n";
10     if( nick1 < nick2 )
11         std::cout << nick1 << " kommt alphabetisch vor " << nick2 << '\n';
12     if( nick2 < nick1 )
13         std::cout << nick1 << " kommt alphabetisch nach " << nick2 << '\n';
14
15     return 0;
16 }
```

► Probieren Sie das Programm aus und beobachten Sie, was passiert.

► Das Prinzip der `if`-Anweisung ist im obigen Quellcode leicht zu begreifen.

- Je nach Ergebnis der jeweiligen Vergleichsoperation (Zeilen 8, 10, 12) wird die direkt folgende Anweisung (Zeile 9 bzw. 11 bzw. 13) ausgeführt oder nicht.



# Vergleichsoperatoren und die `while`-Anweisung

Das Ergebnis von Vergleichsoperationen in Programmen dient meist dazu, den weiteren Ablauf des Programms zu steuern.

## ► Vergleich und `while`-Anweisung:

```
01  int main() {
02      int i1 {0}, i2 {0}, tmp {0}, n {1}; // nur, damit der Code auf die Seite passt
03      std::cout << "Zwei positive ganze Zahlen eingeben: ";
04      std::cin >> i1 >> i2;
05      while( i1 != i2 ) {
06          std::cout << '(' << i1 << ',' << i2 << ')' << std::endl;
07          if( i1 < i2 ) i2 = i2-i1;
08          else { tmp = i1; i1 = i2; i2 = tmp-i1; }
09          n = n + 1;
10      }
11      std::cout << n << " Schritte zum ggT " << i1 << std::endl;
12      return 0;
13  }
```

- Probieren Sie das Programm aus und beobachten Sie, was passiert.
- Das Prinzip der `while`-Anweisung ist im obigen Quellcode leicht zu begreifen.
  - Solange die Vergleichsoperation in der runden Klammer in Zeile 5 `true` ergibt, werden die Anweisungen des nachfolgenden Blocks (Zeilen 6 bis 9) erneut ausgeführt.

# Steuerung des Programmablaufs

Drei Basisprimitive zur Ablaufsteuerung sind hinreichend, um *jedes* Programm zu schreiben.

## ► Sequenz

- Anweisungen werden eine nach der anderen gemäß ihrer programmierten Reihenfolge ausgeführt.
- "Gestapelt" (neben-/hintereinander) bzw. "geschachtelt" (ineinander) im Quellcode.
- *Vorrang* der Operatoren ist bei kombinierten Anweisungen entscheidend (z.B. "Punkt vor Strich"-Regel bei den Grundrechenarten).
  - Im Zweifel immer runde Klammern ( ) setzen.

## ► Selektion

- Anweisungen werden abhängig von Booleschen Bedingungen ausgeführt oder nicht.
- Beispiel `if`-Anweisung.

## ► Iteration

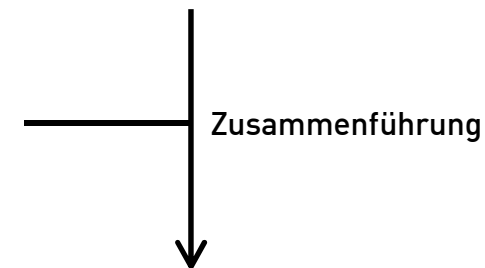
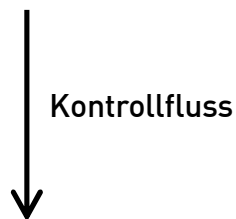
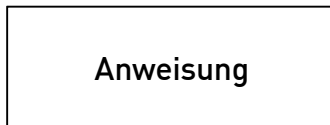
- Anweisungen werden abhängig von Booleschen Bedingungen wiederholt ausgeführt.
- Beispiel `while`-Anweisung.

**Frage:** wenn wir damit schon jedes Programm schreiben können... warum gibt es dann weitere Sprachmittel wie Namensräume, Funktionen oder Klassen?

# Programmablaufplan

Der Programmablauf kann mit einem Programmablaufplan anschaulich dargestellt werden.

- ▶ Programmablaufpläne bestehen aus wenigen, intuitiv verständlichen Symbolen.
- ▶ Modellierungssymbole für Programmablaufpläne:



# Übung

## Programmablaufplan.

- ▶ Zeichnen Sie einen Programmablaufplan für den Quellcode des vorgestellten ggT Programms (S. 17).

# Namensgebung

Namen in C++ sollen mit einem Buchstaben beginnen und dürfen nur Buchstaben, Ziffern und Unterstriche enthalten.

- ▶ Namen werden für Variablen, Typen, Funktionen usw. benötigt, damit man auf sie Bezug nehmen kann.

- ▶ Beispiele für Namen:

- x
- i2
- tmp
- Tmp\_Wert

*Keine* Namen:

2x  
time-2-market  
Hauptmenü  
USD\_to\_EUR

- ▶ Beispiele für gültige, aber nicht unbedingt empfehlenswerte Namen:

- `doube1` - Verwechslungsgefahr mit `double` (nicht für den Compiler, aber für uns Menschen).
- `_foo` - Konfliktgefahr, Namen mit führenden Unterstriche werden üblicherweise von Implementierungs- und Systemelementen verwendet.
- `EIN_NAME` - Namen, die vollständig in Großbuchstaben geschrieben sind, werden üblicherweise für *Makros* eingesetzt, wir wollen Makros *wenn möglich vermeiden*.

# Namensgebung

Suchen Sie sich aussagekräftige und nicht allzu lange Namen, die möglichst schwer zu verwechseln sind.

- ▶ Die von C++ intern reservierten Bezeichner (*Schlüsselwörter* wie `main`) können Sie nicht als Namen für Ihre Variablen, Typen, Funktionen usw. verwenden.
  - Informieren Sie sich selbstständig über die reservierten Schlüsselwörter in einer C++ Referenztabelle.
- ▶ Die Namen von Elementen aus der C++ Standardbibliothek (wie `string` oder `cout`) können Sie verwenden und ihnen lokal neue Bedeutungen geben, sollten das aber tunlichst vermeiden.
- ▶ Sehr kurze Namen (wie `i` oder `tmp`) werden allgemein üblich als lokale Variablen, Parameter oder sog. Schleifenindizes verwendet.
- ▶ Bei längeren, zusammengesetzten Namen setzt man allgemein Unterstriche oder teilweise Großbuchstaben ein (wie `partial_sum` oder `greedyPartition`).
- ▶ Namen sollen so aussagekräftig und intuitiv wie nur möglich sein.
  - **Verwirrende oder kontra-intuitive Namen sind Programmierfehler.**
    - Wie etwa `maxVal` für eine Variable, die nach Programmlogik eben nicht einen "größten Wert" enthält.
- ▶ Namen sollen eine geringe Verwechslungsgefahr haben.
  - Nicht nur Groß- und Kleinschreibung als Unterschied.
  - Vorsicht bei Verwendung leicht verwechselbarer Zeichen wie `o`, `O`, `0`, `1`, `l`, `I`.

# "Ungarische Notation" für Namen

In der sog. "ungarischen Notation" werden Typ und/oder Bedeutung von Namen durch bestimmte Präfixe im Namen selbst gekennzeichnet.

## ► Einfache Beispiele:

bool	bDone	{ false };
const char	ccTab	{ '\t' };
int	iSum	{ 0 };
long double	ldRatio	{ 0.0L };
std::string	strName	{ "IceCap" };

## ► Vorteile, wenn der Typ und/oder weitere semantische Details schon am Namen selbst ersichtlich sind:

- Bei Projekten realer Größe gibt es i.Allg. so viele Namen, dass man nicht auswendig den Überblick behalten kann.
- Ein Name wird in seiner Deklaration mit einem Typ verbunden, und ansonsten im Quellcode ohne wiederholte Nennung seines Typs verwendet.
- Quellcode kann also leichter verständlich sein, wenn der Typ und Hinweise zur Bedeutung bereits am Namen selbst erkennbar sind.
  - Gilt speziell mit überladenen Operatoren, die je nach Typ ihrer Operanden unterschiedlich funktionieren.

## ► Nachteile:

- Zusätzlicher Aufwand.
- Verleitet dazu, die "ungarischen" Präfixe übermäßig zu bewerten und dann die Namen selbst nicht mehr sorgfältig zu wählen.

# Typen und Objekte

Das Konzept der Typen spielt in C++ (und in den meisten anderen, modernen Programmiersprachen) eine zentrale Rolle.

- ▶ Ein *Objekt* ist ein Speicherbereich, in dem ein Wert eines gegebenen Typs abgelegt ist.
- ▶ Ein *Typ* definiert eine Menge von möglichen Werten und einen zugehörigen Satz von Operationen (für ein Objekt).
- ▶ Ein *Wert* ist eine Folge von Bits im Speicher, die entsprechend des Typs verwendet wird.
- ▶ Eine *Variable* ist ein benanntes Objekt.
- ▶ Eine *Deklaration* ist eine Anweisung, die für ein Objekt einen Namen vergibt.
- ▶ Eine *Definition* ist eine Deklaration, die für ein Objekt Speicher reserviert.
- ▶ Eine *Initialisierung* ist eine Anweisung, mit der ein Objekt seinen sinnvollen Anfangswert erhält.



# Objekte: eine anschauliche Vorstellung

Man kann sich ein Objekt bildlich als ein Kästchen vorstellen, in das (nur) Werte des gegebenen Typs hineingelegt werden können.

► `int a { -7 };`

`-7`  
int-Literal

int  
a `-7`

► `char c { 'a' };`

`a`  
char-Literal

char  
c `a`

► `double x { 1.2 };`

`1.2`  
double-Literal

double  
x `1.2`

► `complex<double> z { 1.2, 2.0 };`

complex<double>  
z `1.2` `2.0`

# Objekte: eine anschauliche Vorstellung

Man kann sich ein Objekt bildlich als ein Kästchen vorstellen, in das (nur) Werte des gegebenen Typs hineingelegt werden können.

- Objekte vom Typ `std::string` kennen die Anzahl ihrer Zeichen.

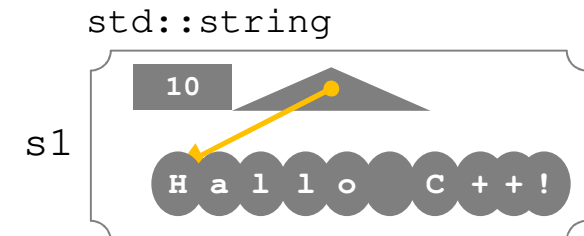
```
std::string s1 { "1.2" };
```

`1.2` \0  
Zeichenketten-Literal



```
s1 = "Hallo C++!";
```

`H a l l o C + + !` \0  
Zeichenketten-Literal



# Typen, Objekte und Speicherplatz

Unterschiedliche Typen von Objekten belegen unterschiedlich viel Speicherplatz.

- ▶ Alle `int` Werte belegen einen gleich großen Speicherplatz.
  - Auf üblichen PCs meist 4 Byte (32 Bit).
- ▶ Alle `bool` und `char` Werte belegen einen gleich großen Speicherplatz.
  - Auf üblichen PCs meist 1 Byte (8 Bit).
- ▶ Alle `double` Werte belegen einen gleich großen Speicherplatz.
  - Auf üblichen PCs meist 8 Byte (64 Bit).
- ▶ Bei `std::string` Werten hängt die Größe des belegten Speicherplatzes von der Länge der Zeichenkette ab.

Die Bits im Speicher werden so verwendet, wie es der Typ des betreffenden Objekts vorsieht.

- Die Bedeutung der Bits im Speicher hängt vollständig von der Interpretation durch den Typ ab.

- Z.B. die 16 höchsten der 64 Bits eines double:

### Bitmuster im Speicher (highest 16, Rest alles 0)

(63-56) (55-48)

00000000 00000000

00000000 00000001

00000000 01100011

```
als: bool
```

false

true

true

char

' \x0 '

' \x1 '

'C'

**int**

0

1

99

double

0.0

0.0625

8.4553e-307

- Die Interpretation der Bits im Speicher kann kompliziert sein.

- Beispiel `double d { 8.4553e-307 };` Interpretation (IEEE 754):

[illegible]

$$(1+\text{man}) * 2^{(\text{exp}-1023)}$$

► Wie im "richtigen Leben", denn was ist (beispielsweise) mit **42** gemeint: Euro, Liter, Jahre, belegte Brote, Zwerge, ...?

# Typsicherheit

C++ ist nicht komplett typsicher.

- ▶ **Typsicherheit** bedeutet, dass Werte im Programm nur so verwendet werden können, wie es der Typ des Objekts vorsieht.
- ▶ In C++ kann man offensichtlich auch Operationen ausführen, die nicht typsicher sind.
  - Die Verwendung einer Variablen vor ihrer Initialisierung ist z.B. nicht typsicher.  
`double x; // Initialisierung vergessen, Wert von x undefiniert`  
`double y { 2.0 + x }; // Anweisung undefiniert`
- ▶ Da C++ typunsichere Sprachmittel kennt,
  - die keine statische Typsicherheit garantieren (d.h. beim Übersetzen des Quellcodes in ein ausführbares Programm durch den Compiler),
  - die keine Prüfung der Typsicherheit zur Laufzeit des Programms vorsehen,müssen wir unsere – deshalb typunsicheren – Programme selbst durch eigene Überprüfungen in unserem Quellcode absichern.

# Sichere Typumwandlungen

In C++ werden bestimmte Typumwandlungen unterstützt, bei denen keine Informationen verloren gehen.

- ▶ Die folgenden Umwandlungen sind sicher (und werden von C++ z.T. implizit durchgeführt):
  - von `bool` nach `char`,
  - von `bool` nach `int`,
  - von `bool` nach `double`,
  - von `char` nach `int`, `char c {'x'}; int i {c}; // i wird zu 120`
  - von `char` nach `double`,
  - von `int` nach `double`, die nützlich ist, aber bei sehr großen `int` Werten (selten, auf einigen Computern) zu Ungenauigkeit führen kann.
- ▶ Bei diesen Umwandlungen werden keine Informationen verloren.
  - Ein Quellwert wird immer in den gleichen Zielwert umgewandelt,
    - bei `double`-Zielwerten in die beste Annäherung.
  - Der Zielwert einer sicheren Umwandlung kann verlustfrei wieder zurück in seinen ursprünglichen Wert umgewandelt werden.

# Unsichere Typumwandlungen

Diese Umwandlungen legen einen Wert in ein Objekt ab, bei dem sich erweisen könnte, dass das Objekt diesen Wert nicht speichern kann.

- ▶ Die folgenden Umwandlungen werden von C++ akzeptiert, obwohl sie unsicher sind (der Compiler warnt allenfalls):

- von double nach int, `double x {2.9}; int i {x}; // i wird 2`
- von double nach char,
- von double nach bool,
- von int nach char, `int i {120}; char c {i}; // c wird 'x'`
- von int nach bool,
- von char nach bool.

- ▶ Der gespeicherte Zielwert kann bei diesen Umwandlungen von dem zugewiesenen Wert abweichen.

```
int i {1234}; char c {i}; // Welchen Wert nimmt c auf Ihrem System an?
```

- ▶ **C++ führt auch unsichere Typumwandlungen ggf. implizit durch.**

# Einige Beispielfragen

Objekte, Typen, Werte, und Steuerungsprimitive.

- ▶ Was ist eine Eingabeaufforderung?
- ▶ Welchen Operator verwenden Sie zum Einlesen eines Werts?
- ▶ Wie könnten Quellcode-Zeilen lauten, mit denen das Programm zur Eingabe einer ganzen Zahl auffordert und diese in einer Variablen namens `number` abspeichert?
- ▶ Wie nennt man `\n` und welchen Zweck hat es?
- ▶ Womit wird in C++ die Eingabe einer `std::string`-Variablen beendet?
- ▶ Was ist ein Objekt?
- ▶ Was ist ein Literal, und welche Arten von Literalen gibt es?
- ▶ Was ist eine Definition? Was ist eine Initialisierung, und wie unterscheidet sie sich von einer Zuweisung?



# Einige Beispielfragen

Objekte, Typen, Werte, und Steuerungsprimitive.

- ▶ Was ist der Unterschied zwischen = und == ?
- ▶ Wie verknüpft man in C++ std::string-Werte?
- ▶ Sehen Sie sich das folgende Quellcode-Fragment an:

```
int i {0}; double d {0.0}; std::string s {" "};  
std::cin >> i >> d >> s;
```

Welche Werte haben i, d und s nach den folgenden Tastatureingaben:

1 .23 Hallo C++	1-2-3 <Strg-z>
1.23 23.009 Hallo	09 0.9 \t
1.23 Hallo	\0x16 2 Hallo

- ▶ Was ist ein Programmablaufplan, und welchen Zweck hat er?
- ▶ Welche Modellierungssymbole für Programmablaufpläne kennen Sie?
- ▶ Welche drei grundlegenden Konstruktionsprimitive zur Ablaufsteuerung von Programmen sind hinreichend, um jedes Programm zu schreiben?

# Einige Beispielfragen

Objekte, Typen, Werte, und Steuerungsprimitive.

- ▶ Nennen Sie je einen C++ Ausdruck für jedes der drei grundsätzlichen Konstruktionsprimitive zur Steuerung des Programmablaufs.
- ▶ Erläutern Sie das Steuerungsprinzip der `if`-Anweisung.
- ▶ Erläutern Sie das Steuerungsprinzip der `while`-Anweisung.
- ▶ Wie könnten Quellcode-Zeilen aussehen, die einer Variablen namens `passed` den Wert `true` zuweisen, falls der Wert einer Variablen `points` größer oder gleich `49.5` ist, und die ansonsten `passed` auf `false` setzen?
- ▶ Wie könnten Quellcode-Zeilen aussehen, die alle 26 Kleinbuchstaben des heutigen lateinischen Alphabets von `a` bis `z` ausgeben?
- ▶ Was bedeutet "Ungarische Notation"?

# Einige Beispielfragen

Objekte, Typen, Werte, und Steuerungsprimitive.

- ▶ Welche der folgenden Namen sind nicht zulässig, und warum nicht?

<code>My_little_lamb</code>	<code>No_1_of_the_Galaxy</code>	<code>2_for_1_offer</code>
<code>latest news</code>	<code>_this_is_ok</code>	<code>number</code>
<code>correct?</code>	<code>string</code>	<code>myString</code>
<code>N01</code>	<code>NO1</code>	<code>O0</code>

- ▶ Finden Sie fünf eigene Beispiele für zulässige Namen, die Sie trotzdem nicht verwenden sollten.
- ▶ Was verstehen Sie unter Typsicherheit? Warum ist sie wichtig?
- ▶ Mit welcher Einheit gibt man die Größe von Speicherbereichen an?
- ▶ Wie groß sind üblicherweise `char`-, `int`- und `double`-Werte auf einem typischen PC?
- ▶ Warum kann die Umwandlung von `double` nach `int` problematisch sein?

## **Nächste Einheit:**

Berechnungen und Anweisungen