2. GRUNDLAGEN I

Inhalt

- Programme und Funktionen
- Datentyp = Werte + Operationen
- Variablen und Zuweisungen
- Ausdrücke und Operatoren
- Kontrollfluss
  - Bedingte Ausführung
  - Wiederholte Ausführung
- Sichtbarkeitsbereich und Lebensdauer
- Konstanten

Programme und Funktionen

```
int ggT(int a, int b) { ... }
```

- Funktionen sind die Grundbausteine von C++-Programmen
  - Jede Funktion hat einen Namen (`ggT`)

    GTK In C++ gilt **declare-before-use**: eine Funktion muss deklariert werden, bevor sie aufgerufen werden kann
  - Eine Funktion kann Argumente erhalten (`a` und `b`)
  - Eine Funktion kann einen Rückgabewert haben (`int`-Wert)

    GTK Wenn eine Funktion keinen Rückgabewert hat, dann ist ihr Rückgabetyp `void`
  - Der **Typ einer Funktion** ist der Typ des Rückgabewerts gefolgt von den Argumenttypen in Klammern (`int(int, int)`)
- Die Ausführung eines C++-Programms beginnt bei der speziellen Methode `main` (vgl. letzte Woche)

Datentyp = Werte + Operationen

- Ein **Datentyp** (kurz: Typ) definiert in Programmiersprachen
  - eine Menge zulässiger Werte, den sogenannten **Wertebereich**
  - **Operationen**, die auf diese Werte angewendet werden dürfen
- Das **Typsystem** einer Programmiersprache definiert
  - **Basistypen** („built-in types“): `bool`, `char`, `int`, `float`, `double` etc.
  - Konstruktoren für **zusammengesetzte Typen**: `[]`, `*`, &, `enum`, `struct`, `class`, `union` etc.
  - Regeln zur **Typisierung** von Ausdrücken
    - **Typinferenz**: wie wird der Gesamttyp eines Ausdrucks (automatisch) bestimmt?
    - **Typumwandlung**: welche Ausdrücke können explizit oder implizit vom einen Typ zum anderen umgewandelt werden?
    - **Zuweisungskompatibilität**: welche Ausdrücke können einander zugewiesen, miteinander verglichen oder miteinander verknüpft werden?

Ganze Zahlen

- Basistyp `int` für **ganze Zahlen** ist typischerweise 32 Bit groß
- Die **Vorzeichenhaftigkeit** und **Größe** können modifiziert werden
  - `signed`/`unsigned`
  - `short`/`long`/`long long`
- GTK Wenn Modifikatoren verwendet werden, kann das Schlüsselwort `int` auch weggelassen werden

Typspezifikation	Typischer Wertebereich
`short int`	$-2^{15} \dots 2^{15} - 1$
`int`	$-2^{31} \dots 2^{31} - 1$
`long int`	$-2^{63} \dots 2^{63} - 1$
`unsigned`	$0 \dots 2^{32} - 1$
`unsigned long int`	$0 \dots 2^{64} - 1$

Ganze Zahlen

- C++ definiert die folgenden Syntax für `int`-**Literale**
  - Dezimale Schreibweise (Basis 10)
  - Oktale Schreibweise (Basis 8)
  - Hexadezimale Schreibweise (Basis 16)
  - Binäre Schreibweise (Basis 2)
- Suffixe für Vorzeichenhaftigkeit und Größe
  - `u` oder `U` für `unsigned`
  - `l` oder `L` für `long`
- GTK Literale können durch Apostrophe strukturiert werden

```
int d = 1701;              // Basis 10
int o = 03245;             // Basis  8
int x = 0x6a5;             // Basis 16
int b = 0b11010100101;     // Basis  2
```

Zeichen

- Basistyp `char` für **Zeichen** ist 8 Bit groß: $-128 \ldots 127$
  - Speichert ein ASCII-Zeichen
  - Kann auch mit `unsigned` modifiziert werden: $0 \ldots 255$
- C++ definiert die folgende Syntax für `char`-**Literale**
  - Zeichenkonstanten werden durch Apostrophe eingefasst
  - Nichtdruckbare Zeichen und Sonderzeichen werden mithilfe des \ („escape“, „backslash“) eingegeben
    - Dezimaler, oktaler oder hexadezimaler ASCII-Code
    - Vordefinierte Abkürzung

```
char a = 'a';           // Kleinbuchstabe "a"
char b = '\x62';        // Kleinbuchstabe "b"
char n = '\n';          // Zeilenumbruch (new line)
char s = '\\';          // Backslash
char q = '\'';          // Apostroph
```

Fließkommazahlen

- Für **IEEE-754 Fließkommazahlen** gibt es in C++ zwei Basistypen
  - `float` ist 32 Bit groß
  - `double` ist 64 Bit groß
  - **GTK** In der Regel verwendet man `double`, außer es müssen sehr viele solcher Zahlen gespeichert (Platz) oder sehr viele Operationen ausgeführt (Geschwindigkeit) werden
- Syntax der **Literale** für Fließkommazahlen
  - Können dezimal oder hexadezimal geschrieben werden
  - Müssen einen Dezimalpunkt oder einen Exponenten beinhalten
  - Suffix `f` oder `F` für `float`

```
double d = 1E10;
double pi = 3141.5926e-3;
float e = 2.71828f;
```

Wahrheitswerte

- Basistyp `bool` für **Wahrheitswerte** ist typischerweise 8 Bit groß, hat aber nur zwei Werte: `true` (wahr) und `false` (falsch)
- Boolesche Ausdrücke werden fast ausschließlich dazu verwendet, den Kontrollfluss zu steuern
  - bedingte Anweisungen
  - Schleifen
- Analog zu arithmetischen Ausdrücken kann man aber auch Boolesche Ausdrück formulieren und sie Variablen zuweisen

```
bool flag = true;
bool ungerade = (27 % 2 == 1);
bool gerade = !ungerade;
```

Zeichenketten

- Es gibt mindestens zwei Möglichkeiten, um mit **Zeichenketten** („Strings“) in C++ zu arbeiten
  - `char[]` stellt den String als ein Array von Zeichen dar
    - tiefe Abstraktionsebene $\Rightarrow$ viel Handarbeit
    - Null-Terminierung: String endet mit dem Zeichen `'\0'`
    - Es gilt: Länge des Arrays $>$ Länge des Strings
    - **GTK** Diese Art von String wird als „C-String“ bezeichnet
  - `std::string` ist Teil der C++-Standardbibliothek
    - höhere Abstraktionsebene $\Rightarrow$ weniger Handarbeit
    - Wenn immer möglich, verwenden wir diese Art von String!
- C++ definiert folgende Syntax für String-**Literale**
  - String-Literale werden durch doppelte Anführungszeichen eingefasst
  - Nichtdruckbare Zeichen und Sonderzeichen werden wiederum mithilfe des `\` eingegeben

Variablendeklaration

- In C++ muss jede Variable **vor** Gebrauch deklariert werden.

```
std::string text;
```

- Eine **Variablendeklaration** besteht aus dem **Namen** (`text`) und dem **Datentyp** (`std::string`) der Variablen
  - Der Variablenname ist eine Abstraktion für den **Variablenwert**, der an einer gewissen Speicheradresse steht
  - Der Datentyp legt fest, wie viel **Speicherplatz** benötigt wird, und hilft, Fehler beim Umgang mit den Werten zu erkennen.

- **GTK** Selbsterklärende Variablennamen sind besser als kryptische Abkürzungen!
  - Länge des Namens hat keinen Einfluss auf die Laufzeit
  - Kombination von Worten mit **camelCase** oder **Unterstrich**

Variablendefinition

- Wird einer Variablen bei der Deklaration ein Wert zugewiesen, spricht man von einer **Variablendefinition**
  - Typischerweise ist dieser Wert ein Literal
  - Solche Zuweisungen nennt man auch **Initialisierung**
- Andernfalls hat die Variable einen beliebigen, unbekannten Wert!
  - Ausnahmen bestätigen die Regel: `std::string`, `std::vector` etc.
- C++ bietet verschiedene Schreibweisen für Initialisierung an
  - Gewöhnliche Zuweisung (=)
  - Initialisierungslisten mit geschweiften Klammern `{...}`

```
double d1 = 3.1;      // Initialisiert d1 mit 3.1
double d2 {3.1};      // Initialisiert d2 mit 3.1
double d3 = {3.1};    // Initialisiert d3 mit 3.1
```

Ey Mann, wo is' mein `auto`?

- Wenn sich der Typ einer Variablen aus ihrer Definition ableiten lässt, dann muss er **nicht explizit** angegeben werden.
  - Schlüsselwort `auto` statt des Typnamens verwenden
  - Mit `auto` verwendet man typischerweise = anstatt `{}`
  - `auto` macht den Code häufig lesbarer und kürzer
- GTK Das automatische Ableiten von Typen wird als **Typinferenz** („type inference“) bezeichnet.

```
auto b = true;     // b hat den Typ bool
auto d = 3.141;    // d hat den Typ double
auto i = 42;       // i hat den Typ int
auto z = sqrt(d); // z hat den Typ des Ergebniswerts
                   // des Aufrufs sqrt(d)
```

Zuweisung

Wir haben bereits beispielhaft gesehen, wie Werte in Variablen gespeichert werden. Nun führen wir die **Zuweisung** („assignment“) noch formal ein.

Zuweisung

Der **Zuweisungsoperator** = ist ein binärer Operator. Auf seiner linken Seite steht die zu belegende Variable. Auf der rechten Seite steht der Ausdruck, dessen Wert die Variable erhalten soll. Der Zuweisungsoperator macht zwei Dinge.

1. Er prüft, ob der Typ des Werts mit dem Typ der Variable **kompatibel** ist
2. Ist dies der Fall, **kopiert** er den Wert und speichert ihn in der Variablen

Zuweisung

- Normale Zuweisung

```
umfang = 2 * 3.14159 * radius;
```

  - Auf der linken Seite **muss** eine Variable stehen
  - Auf der rechten Seite steht ein **Ausdruck**

- Abkürzungen für häufige Muster von Zuweisungen

```
++i;       // Identisch zu i = i + 1;
--i;       // Identisch zu i = i - 1;

x += 3;    // Identisch zu x = x + 3;
x -= 3;    // Identisch zu x = x - 3;
x *= 3;    // Identisch zu x = x * 3;
x /= 3;    // Identisch zu x = x / 3;
x %= 3;    // Identisch zu x = x % 3;
```

Ausdrücke

Wir haben den Begriff **Ausdruck** schon mehrmals verwendet, ohne ihn genau zu definieren. Das holen wir nun nach!

Ausdücke

Ein Ausdruck („expression“) repräsentiert einen Wert, den man erhält, indem der Ausdruck ausgewertet wird. Ein Ausdruck besteht aus Operanden („operands“) und Operatoren („operators“).

- Operanden sind Literale, Variablen, Rückgabewerte von Methodenaufrufen oder Teilausdrücke.
- Operatoren sind datenspezifische Operationen, die auf Operanden angewendet werden, um neue Werte zu bilden.

Operatoren

Die folgende Tabelle gibt einen (unvollständigen) Überblick über die Operatoren in C++.

Kategorie	Syntax	Semantik
Vorzeichen	`+ -`	Positive und negative Zahlen
Arithmetik	`+ - * / %`	Arithmetische Grundrechenarten (inklusive Modulo)
Logik	`! && ||`	Operationen der Boolschen Algebra
Vergleiche	`== != > >= < <=`	Vergleiche von Werten und Ausdrücken
Drei-Wege-Operator	`?:`	Bedingte Ausdrücke
Bitweise Operatoren	`& | ^ ~ << >>`	Operationen auf der Bitebene
Zuweisung	`=`	Variablenzuweisung
	`+= -= *= /= %=`	... in Kombination mit den arithmetischen Operatoren
	`&= |= ^= ~= <<= >>=`	... in Kombination mit den bitweisen Operatoren
	`++ --`	Inkrement und Dekrement

Operatorenarten

Unäre Operatoren werden auf genau **einen** Operanden angewendet

- Positives (+) und negatives (−) Vorzeichen
- Logische Negation (`!`)
- Bitweises NOT (~)
- Inkrement (++) und Dekrement (−−)

Binäre Operatoren werden auf genau **zwei** Operanden angewendet

- Arithmetische Operationen (`+ - * / %`)
- Logische Konjunktion (&&) und Disjunktion (||)
- Bitweises AND (&), OR (|), XOR (^), LSH (<<) und RSH (>>)
- Vergleiche (`== != > >= < <=`)
- Zuweisungen

Ternäre Operatoren werden auf genau **drei** Operanden angewendet

- Drei-Wege-Operator (`?:`)

Operatorpräzedenz: „Punkt vor Strich, u.s.w.“

Präzedenz	Operator	Beschreibung
2	`a++ a--`	Inkrement und Dekrement (Postfix)
3	`++a --a`	Inkrement und Dekrement (Prefix)
	`+a -a`	Positives und negatives Vorzeichen
	`! ~`	Logische Negation (NOT) und bitweises NOT
5	`a*b a/b a%b`	Multiplikation, Division und Modulo
6	`a+b a-b`	Addition und Subtraktion
7	`<< >>`	Bitweiser Links- und Rechts-Shift
9	`< <= > >=`	Vergleichsoperatoren größer (gleich) und kleiner (gleich)
10	`== !=`	Vergleichsoperatoren gleich und ungleich
11	`a&b`	Bitweises AND
12	`^`	Bitweises XOR (exklusives OR)
13	`|`	Bitweises OR (inklusives OR)
14	`&&`	Logische Konjunktion (AND)
15	`||`	Logische Disjunktion (OR)
16	`a?b:c`	Drei-Wege-Operator
	`=`	Zuweisungen (alle Formen)

Zuweisungen sind Ausdrücke

Die folgenden Anweisungen sind gültiges C++.

```
int i = 1;
int j = (i = 5) * i;
```

- Das Ergebnis des Ausdrucks, der `j` zugewiesen wird, ist allerdings **nicht definiert**
  - Nach der Ausführung kann `j == 5` oder `j == 25` gelten
  - Ergebnis hängt vom verwendeten Compiler und Optimierungen ab
- **GTK** Solche Konstrukte gilt es zu vermeiden!
  - Programme werden nicht-deterministisch und damit fehlerhaft
  - Auch g++ ist mit solchem Programmcode gar nicht zufrieden

```
warning: unsequenced modification and
access to 'i' [-Wunsequenced]
```

Numerische Promotion und Konversion

Der Compiler übersetzt alle Operatoren in CPU-Instruktionen. Diese Instruktionen erwarten, dass alle Operanden vom **gleichen** Typ sind.

Haben Operanden **unterschiedliche** Typen, werden die Werte **implizit** auf den **spezifischsten gemeinsamen** Typ gebracht.

Dieser gemeinsame Typ ist auch der Ergebnistyp des Operators.

Numerische Promotion („numerical promotion“)

- Operanden sind alles ganze Zahlen **oder** alles Fließkommazahlen
- Der größte verwendete Datentyp ist der gemeinsame Typ
- Kleineren Werte werden zu größeren Werten **erweitert**

Numerische Konversion („numerical conversion“)

- Operanden sind ganze Zahlen **und** Fließkommazahlen
- Der größte der Fließkommazahltypen ist der gemeinsame Typ
- Ganzzahligen Werte werden in Fließkommawerte **umgewandelt**

Numerische Promotion und Konversion

- `auto a = 3 * 5;`
  - Beide Operanden sind vom gleichen Typ (`int`)
  - Somit können die Operanden direkt multizipliert werden
  - Der Typ von `a` ist deshalb auch `int`
- `auto b = 3 * 5l;`
  - Der erste Operand ist ein `int`, der zweite ein `long`
  - Der erste Operand wird zu einem `long` **erweitert**
  - Das Ergebnis der Multiplikation und der Typ von `b` sind `long`
- `auto c = 3.0 * 5;`
  - Der erste Operand ist ein `double`, der zweite ein `int`
  - Der zweite Operand wird in einen `double` **umgewandelt**
  - Das Ergebnis der Multiplikation und der Typ von `c` sind `double`

Bedingte Ausführung von Code

Mit der **`if`-Anweisung** wird Code nur dann ausgeführt, wenn eine gewisse Bedingung erfüllt ist.

```
if (alter <= 16) {
    eintrittspreis = 16.0;
} else if (alter >= 65) {
    eintrittspreis = 18.0;
} else {
    eintrittspreis = 21.0;
}
```

- Die Bedingungen werden von oben nach unten geprüft
  - Falls eine Bedingung wahr ist, wird der entsprechende Code-Block ausgeführt
  - Weitere Bedingungen werden dann **nicht** mehr geprüft
  - Ist keine Bedingung erfüllt, wird der **`else`**-Teil ausgeführt
- Es kann beliebig viele (auch keine) **`else-if`**-Teile, aber nur maximal einen **`else`**-Teil geben

Bedingte Ausführung von Code

Das `switch`-**Statement** ist eine gute Wahl bei vielen einfachen Gleichheitsbedinungen.

```
int x = 0, y = 0;
switch (ch) {
  case 'n': ++y; break;    // north, move up y-axis
  case 's': --y; break;    // south, move down y-axis
  case 'e': ++x; break;    // east, move up x-axis
  case 'w': --x; break;    // west, move down x-axis
  default: ...;            // handle unknown command
}
```

- Der `default`-Teil ist **optional**
  - Weglassen kann aber zu unbemerkten Fehlern führen
  - Deshalb warnen Compiler oder Stil-Checker häufig davor
- GTK Ohne `break` werden auch alle anschließenden Fälle ausgeführt!

Bedingte Ausführung von Code

Mit dem sogenannten **Drei-Wege-Operator** können einfache `if-else`-Anweisungen in einer Zeile geschrieben werden.

```
auto maximum = x > y ? x : y;
```

- Ist die Bedingung (`x > y`) wahr, ist der Wert des gesamten Ausdrucks der Wert des ersten Teilausdrucks (`x`) und sonst des zweiten Teilausdrucks (`y`)
- Beide Teilausdrücke (`x` und `y`) müssen vom **gleichen** Typ sein

Programmieren!

- Schreibe ein C++-Programm `calendar.cpp`, das
  - ein Datum in Form von drei Kommandozeilenargumenten einliest,
  - mit Zellers Kongruenz den entsprechenden Wochentag berechnet und
    $$h = \left(q + \left\lfloor \frac{(m+1)\cdot 13}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2 \cdot J\right) \mod 7$$
  - diesen als Text auf der Konsole ausgibt.
- Beispielaufruf

```
> g++ -o calendar calendar.cpp
> ./calendar 31 10 2023
Dienstag
```

Wiederholte Ausführung von Code

Mit **Schleifen** kann ein Code-Block wiederholt ausgeführt werden.

- Vorprüfende oder kopfgesteuerte Schleifen
  - Der Code-Block wird **beliebig oft** (inkl. kein mal) ausgeführt
  - `while`-Schleife
  - `for`-Schleife
- Nachprüfende oder fußgesteuerte Schleifen
  - Der Code-Block wird **mindestens einmal** ausgeführt
  - `do`-Schleife

Kopfgesteuerte Schleifen

- Gib die Zahlen von 100 bis 1 mit einer **`while`-Schleife** aus

```
auto i = 100;
while (i > 0) {
  std::cout << i << std::endl;
  --i;
}
```

- Äquivalent dazu, aber kürzer und lesbarer mit einer **`for`-Schleife**

```
for (auto i = 100; i > 0; --i) {
  std::cout << i << std::endl;
}
```

- **GTK** Konvention zur Verwendung von **`while`**- und **`for`**-Schleifen
  - **`for`**-Schleife nur bei einer Schleifenvariablen und (relativ) einfacher Abbruchbedingung
  - **`while`**-Schleife in allen anderen Fällen

Fußgesteuerte Schleifen

- Gib die Zahlen von 100 bis 1 mit einer **do-Schleife** aus

```
auto i = 100;
do {
  std::cout << i << std::endl;
  --i;
} while (i > 0);
```

- GTK Fußgesteuerte Schleifen sind in der Praxis seltener als kopfgesteuerte Schleifen

Schleifen- und Iterationsabbruch

- Manchmal ist es nützlich, die Ausführung der Schleife nicht nur über die Schleifenvariable und die Schleifenbedingung zu steuern
  - `break` bricht die **aktuelle Schleife** ab
    $\Rightarrow$ die Ausführung geht hinter der Schleife weiter
  - `continue` bricht die **aktuelle Iteration** ab
    $\Rightarrow$ die Ausführung geht in der nächsten Iteration weiter

```
do {
  auto code = std::cin.get();    // liest ein Zeichen ein
  if (code == 27) break;         // ASCII-Code von Escape
  if (code < 'a' || code > 'z') continue;
  std::cout << "Code:␣" << code << std::endl;
} while (true);
```

- GTK Bei geschachtelten Schleifen wird nur die Schleife, in der das `break` steht, abgebrochen, nicht aber die umgebenden Schleifen!

Programmieren!

- Schreibe ein C++-Programm `factorial.cpp`, das
  - eine ganze Zahl (n) von der Kommandozeile einliest,
  - die Fakultät ($n!$) dieser Zahl berechnet und
  - das Ergebnis auf der Kommandozeile ausgibt.
- Beispielaufruf

```
> g++ -o factorial factorial.cpp
> ./factorial 5
120
```

Sichtbarkeitsbereich und Lebensdauer

- Eine Deklaration führt ihren Namen in einen sogenannten **Sichtbarkeitsbereich** („scope“) ein
- C++ unterscheidet verschiedene Scopes
  - Local Scope (mehr dazu gleich)
  - Class Scope (mehr dazu in Woche 6 und 7)
  - Namespace Scope (mehr dazu in Woche 5)
  - Global Scope (mehr dazu in Woche 5)
- Scopes erfüllen mehrere wichtige Aufgaben
  - **Auflösung von Namen** („name resolution“): Auf welche Deklaration bezieht sich die Verwendung eines Names?
  - **Lebensdauer** („lifetime“): Wann soll Speicher belegt und wann kann er wieder freigegeben werden?

Sichtbarkeitsbereich und Lebensdauer

Local Scope

Wird ein Name innerhalb einer Funktion deklariert, so sprechen wir von einem **lokalen Namen** („local name“).

- Der Sichbarkeitsbereich eines lokalen Namens erstreckt sich vom Punkt der Deklaration bis ans Ende des Blocks
- Ein Block ist durch ein `{ }` Paar begrenzt
- Die Namen von Funktionsargumenten gelten als lokale Namen

Sichtbarkeitsbereich und Lebensdauer

```
1  void scopes(int p) {
2    int i = p * j; // Fehler!
3    int j = 42;
4    for (int j = 0; j < p; j++) {
5      std::cout << j << std::endl;
6    }
7    std::cout << j << std::endl;
8  }
```

- Der Scope von `p` erstreckt sich von Zeile 1 bis Zeile 8
- Da der Sichtbarkeitsbereich von `j` erst in Zeile 3 beginnt, kann `j` in Zeile 2 noch nicht verwendet werden
- Das `j` **in** der `for`-Schleife (Zeile 5) bezieht sich auf die Deklaration in Zeile 4
- Das `j` **nach** der `for`-Schleife (Zeile 7) bezieht sich auf die Deklaration in Zeile 3

Konstanten

In C++ gibt es zwei Arten der **Unveränderbarkeit**.

- `const` „Ich verspreche, diesen Wert nicht zu verändern“
  - Der Compiler prüft und erzwingt dieses Versprechen
  - Der Wert darf zur Laufzeit berechnet werden
- `constexpr` „Der Wert soll zur Compile-Zeit berechnet werden“
  - Spezifiziert eine schreibgeschützt gespeicherte Konstante
  - Der Wert muss zur Compile-Zeit berechnet werden können

```
constexpr double square(double x) { return x * x; }

int v = 17;
const int c = 17;

constexpr double e1 = 1.4 * square(v);    // Fehler
const double e2 = 1.4 * square(v);        // Okay
constexpr double e3 = 1.4 * square(c);    // Okay
```