Rechtlicher Hinweis

Legal notice

Informatik I (B.Sc. Physik)

Funktionen in C++

Dr. Paul Bodesheim
(Paul.Bodesheim@uni-jena.de)

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Fakultät für Mathematik und Informatik
Lehrstuhl für Digitale Bildverarbeitung

SoSe 2020

Inhalt

Inhalt

Funktionen in der Programmierung allgemein

- Idee/Ziel: Wiederverwendbarkeit von Code
  - Programmteil mehrfach im Programm verwenden

    (an unterschiedlichen Stellen, sodass Schleife nicht funktioniert)
  - Programmteil in anderen Programmen verwenden
  - Beispiele: Nutzereingaben, Ausgabeformatierung, wiederkehrende Berechnungen, ...

- Übliche Bezeichnungen:
  - Unterprogramme
  - Subroutinen
  - Prozeduren
  - Funktionen

Funktionen vs. Prozeduren

- Übliche Unterscheidung/Bezeichnung von Unterprogrammen:
  - Funktion: liefert (Funktions-) Wert zurück

    $\Rightarrow$ Beispiel: a=sin(x);
  - Prozedur: liefert keinen Wert zurück

    $\Rightarrow$ Beispiel: print (x);

- Sinnvolle Unterscheidung durch Kennzeichnung ihrer Anwendung:
  - Funktionen werden in Ausdrücken aufgerufen bzw. überall dort, wo ein Wert erwartet wird
  - Prozeduren stehen "für sich", sind normale Anweisungen (sind Synonym/Alias für eine Menge von Anweisungen)

- C++ kennt nur Funktionen: alle Unterprogramme haben Rückgabewert
- Typ der Funktion = Typ des Rückgabewertes

Inhalt

Rückgabewerte von Funktionen

- Spezifikation des Rückgabewertes in der Definition einer Funktion durch Angabe des Datentyps für den Rückgabewert
  - Alle Datentypen sind möglich: **int** , **double**, string , ...
  - Es gibt immer genau einen Rückgabe-Datentyp!
  - (Es gibt Möglichkeiten, mehrere Werte zurückzugeben. Später mehr dazu!)
- Spezieller Rückgabe-Typ:
  - Schlüsselwort: **void**
  - Signalisiert, dass nichts zurückgegeben wird
  - Funktionen mit Rückgabe-Typ **void** sind eigentlich Prozeduren
- Achtung: auch Funktionen mit anderem Rückgabe-Datentyp lassen sich als Prozeduren verwenden, der Rückgabewert wird dabei einfach ignoriert

Definition von Funktionen in C++

```
<TYP> <NAME> ( <PARAMETERLISTE> )
{
    <FUNKTIONSRUMPF>
}
```

- <TYP>: Datentyp des Rückgabewertes der Funktion (Typen für Variablen) oder "spezieller Typ" **void**
- <NAME>: Name der Funktion, mit allen Bedingungen/Regeln für Namen (Bezeichner) in C++
- <PARAMETERLISTE> in *runden* Klammern: Liste von Variablen, deren Werte an die Funktion übergeben werden (kann auch leer sein)
- <FUNKTIONSRUMPF> in *geschweiften* Klammern: auszuführende Anweisungen der Funktion als Block (beachte Gültigkeit lokaler Variablen)

return-Anweisung

- Schlüsselwort: **return**
- Gefolgt von einem Ausdruck, dessen Wert vom spezifizierten Datentyp ist
- Legt Rückgabewert fest
- Beendet die Funktion und veranlasst Rückkehr zum Funktionsaufruf (Hauptprogramm)

- Beispiele:
  - **return** 0; // Literal
  - **return** x; // Variable
  - **return** y+z; // arithmetischer Ausdruck
  - **return** **true**; // Wahrheitswert

- Auch möglich: **return**;

  ⇒ Vorzeitiger Abbruch einer Funktion des Typs **void**

  ⇒ Ignoriert folgende Anweisungen und verlässt Funktion

Beispiele

```
// Funktion (Prozedur), die eine Trennlinie ausgibt
void printLine()
{
    for (int i=0; i<80; i++)
        cout << "-";
    cout << endl;
}

// Funktion zur Eingabe einer nicht-negativen Zahl
int inputNonnegative()
{
    int zahl;
    cout << "Bitte eine nicht-negative Zahl eingeben: ";
    cin >> zahl;

    while (zahl < 0)
    {
        cout << "Die Eingabe ist negativ, versuche es erneut: ";
        cin >> zahl;
    }

    return zahl;
}
```

Beispielaufrufe im Hauptprogramm

```
int main()
{
    printLine();

    int zahl1;
    zahl1 = inputNonnegative();

    int zahl2 = inputNonnegative();
    int produkt = zahl1*zahl2*inputNonnegative();

    cout << "Das Produkt der 4 Zahlen ist: " <<
        produkt*inputNonnegative() << endl;

    inputNonnegative(); // Eingegebene Zahl geht verloren!

    return 0;
}
```

- Auch das Hauptprogramm main ist eine Funktion (Hauptfunktion, main-Funktion)!
- Wird vom System beim Start des übersetzten Programms aufgerufen
- Verwendung von Rückgabewert und Parametern der Hauptfunktion später!

Inhalt

Parameter einer Funktion

- Parameter können einer Funktion genauer mitteilen, was zu tun ist

  $\Rightarrow$ Flexibilität, Berechnungen/Anweisungen abhängig von Parameterwerten
- Datentyp gefolgt von (lokalem) Parameternamen
- Beispiel 1: allgemeine Eingabe mit konfigurierbarem Aufforderungstext

```
int inputGeneral(string text)
{
    int zahl;
    cout << text;
    cin >> zahl;
    return zahl;
}
```

- Beispiel 2: Trennlinie mit variabler Länge

```
void printLine(int length)
{
    for (int i=0; i<length; i++)
        cout << "-";
    cout << endl;
}
```

Parameterliste

- Einer Funktion können auch mehrere Parameter übergeben werden
- Auflisten von mehreren Parametern mit zugehörigem Datentyp
- Mehrere Parameter durch Komma getrennt

- Ähnlich zur Definition von Variablen, allerdings können nicht mehrere Parameter auf einmal definiert werden

  $\Rightarrow$ Jeder Parametername benötigt Angabe eines Datentyps!

- Beispiele:
  - **int** sum3(**int** a, **int** b, **int** c)
  - **void** printText(string text, **int** numRepetitions)

Werte und Variablen

- Die Begriffe **Wert** und **Variable** im Kontext der Parameter anhand der Zuweisung:

```
                y = 4 * m + n;
<VARIABLE> = <WERT>;
<VARIABLE> = <AUSDRUCK>;
  <lvalue> = <rvalue>;
```

- Zunächst wird der **Wert** auf der rechten Seite berechnet und dann der **Variablen** auf der linken Seite zugewiesen
- Besser **rvalue** und **lvalue**, um den Begriff **Variable** zu vermeiden
- **Variable**: durch Namen gekennzeichneten Speicherplatz im Rechner, der Werte (eines Typs) aufnehmen kann
- Unterscheidung bei den Parametern einer Funktion:
  - Soll der Funktion ein **Wert** übergeben werden (z.B. eine Zahl, aus der die Wurzel berechnet wird)?
  - Oder soll eine **Variable** übergeben werden, also ein Speicherplatz, in dem das Ergebnis einer Berechnung oder Eingabe abgelegt werden kann?

  $\Rightarrow$ Wert-Parameter vs. Variablen-Parameter

Inhalt

Wert-Parameter

Soll einer Funktion ein Wert übergeben werden, so ist in die Parameterliste ein Typ und ein formaler Name (Bezeichner) für den Parameter einzutragen.

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

double distance(double x1, double x2)
{
    return fabs(x1-x2);
}
```

- Zugriff auf übergebenen Wert mit Parametername
- Wert-Parameter verhalten sich wie lokale Variablen (Initialisierung mit übergebenem Wert bei Funktionsaufruf, Lebensdauer bis zum Ende der Funktion)
- Wert des Parameters darf innerhalb der Funktion überschrieben werden
- Lokale Variable der Funktion ändert nichts im aufrufenden (Haupt-) Programm

Beispiel: Werte-Parameter

```
int plus1(int x)
{
    x = x+1;
    return x;
    // oder direkt return x+1;
}

int main()
{
    int x = 4;
    int y = plus1(x);
    cout << x << endl; // 4
    cout << y << endl; // 5
}
```

- Wichtig: Werte werden beim Funktionsaufruf kopiert!
- Wert der Variablen x im Hauptprogramm wird bei Funktionsaufruf kopiert und einer weiteren, lokalen Variable zugewiesen
- Diese hat hier den gleichen Namen, aber einen anderen Gültigkeitsbereich (Überdeckung)

Technische Erläuterungen zu Werte-Parametern

- Wie wird ein Wert-Parameter übergeben?
- Ein Wert repräsentiert keinen Speicherplatz, er muss zur Verarbeitung aber irgendwo im Speicher abgelegt werden
- Er wird auf den Stack bzw. einen Speicherplatz des Stacks geschrieben (**temporärer** Speicherplatz)
- Dieser Speicherplatz steht in der Funktion zur Verfügung und kann mit dem Parameternamen angesprochen werden
- Analog wird bei der Wert-Rückgabe mittels **return** der Rückgabewert auf einen temporären Speicherplatz geschrieben/kopiert, sodass er darüber im aufrufenden Hauptprogramm verwendet werden kann

Inhalt

Variablen-Parameter bzw. Referenz-Parameter

- Soll einer Funktion eine Variable übergeben werden, verwendet man Referenz-Parameter
- Referenzen im Detail später, wenn es um Zeiger (Pointer) geht
- Referenzen sind in C++ “Aliasnamen” für Variablen
- Eine Referenz kann genauso verwendet werden wie die Variable, für die die Referenz steht
- Kennzeichnung einer Referenz durch &-Symbol zwischen Datentyp und Name

```
void inc(int & a)
{
    a = a+1; // oder a++;
}

int main()
{
    int x = 4;
    inc(x);
    cout << x << endl; // 5
}
```

Erläuterungen zu Referenz-Parametern

- Weitere Form, um in der Funktion berechnete Werte "zurückzugeben"
- Keine Rückgabe im eigentlichen Sinne, da Variable (Speicherplatz) aus dem aufrufenden Programmteil direkt verändert wird (nur über anderen Namen: Alias)
- Mit Referenz-Parametern lassen sich mehrere Rückgabewerte (Funktionswerte) einer Funktion realisieren
- Prozeduraler Charakter mit Rückgabetyp **void**
- Beim Funktionsaufruf werden keine Werte kopiert
- Gleiches gilt bei "Rückgabe" eines Funktionswertes über Referenz-Parameter
- Anlegen einer Referenz erstellt nur neuen Namen (Alias), fordert keinen zusätzlichen Speicherplatz an

  $\Rightarrow$ schneller und speichereffizienter (vor allem bei komplexeren Datentypen: string bzw. Arrays/Vektoren, Records/Strukturen, Objekte)

Beispiel: Tauschen von Werten

```
void swap(int & a, int & b)
{
    int h = a;
    a = b;
    b = h;
}

int main()
{
    int x = 4;
    int y = 7;
    swap(x,y);
    cout << x << endl; // 7
    cout << y << endl; // 4
}
```

Inhalt

Konstante Referenz-Parameter

- Auch "konstante Referenzen" genannt
- Kombination der Vorteile von den beiden vorangegangenen Parameter-Varianten (Werte- und Referenz-Parameter)
- Mit einem konstanten Referenz-Parameter kann man:
  - Einen Wert übergeben (auch den Wert einer Variablen)
  - Eine Kopie vermeiden
  - Den aktuellen Wert des Parameters im aufrufenden Programm unverändert lassen
- Im Prinzip: Referenz-Parameter, dessen Wert in der Funktion nicht verändert werden darf (keine Wertzuweisung erlaubt)
- Entweder: durch Disziplin (fehleranfällig, Gefahr des Vergessens)
- Oder: Referenz-Parameter als konstant definieren
- Schlüsselwort: **const** (vor Datentyp)
- Compiler meldet Fehler, wenn innerhalb einer Funktion der Wert einer konstanten Referenz verändert wird

Beispiel für konstante Referenz-Parameter

```
int add1(const int & a)
{
    // a = a+1; funktioniert nicht mehr!
    return a+1;
}
```

- Hauptanwendung: wenn Kopie viel Platz oder Zeit benötigt

  $\Rightarrow$ besser als Wert-Parameter

- Im Beispiel kein Unterschied/keine Relevanz, da Platzbedarf und Kopieraufwand einer Integer-Variable klein ist

- Bei Zeichenketten (string) lohnt es sich schon, ebenso später bei komplexeren Datentypen

Vergleich der Parameter-Varianten

- Was ist beim Funktionsaufruf f(z); für die verschiedenen Parameter-Varianten erlaubt?
- Was darf man verwenden?

z ist ...	Wert-Parameter **void** f(**int** z)	Referenz-Parameter **void** f(**int** & z)	Konstanter Referenz-Parameter **void** f(**const int** & z)
Konstante, z.B. 5	ja	nein	ja
Variable, z.B. x	ja	ja	ja
Ausdruck, z.B. $4 * x + 1$	ja	nein	ja

Inhalt

Signatur einer Funktion

- Bestandteile des Funktionskopfes bei der Funktionsdefinition:
  <Typ> <Name> (<Parameterliste>)

- Signatur: Funktionsname, Anzahl und Typ der Parameter (ohne Rückgabetyp!)
- **double** square(**double** a) ⇒ Signatur: square(**double**)
- **int** sum3(**int** a, **int** b, **int** c) ⇒ Signatur: sum3(**int**,**int**,**int**)

- Für die korrekte Zuordnung beim Aufruf einer Funktion ist die Signatur entscheidend ⇒ Funktionen müssen sich in C++ in der Signatur unterscheiden
- Compiler entscheidet anhand der Signatur, welche die richtige Funktion ist

- Hinweis: in C musste der Funktionsname eindeutig sein
  int abs(**int** a) vs. **double** fabs(**double** a)

Überladen von Funktionen

- Solange unterschiedliche Funktionsnamen verwendet werden, ist auch die Signatur verschieden
- Bei exakt gleicher Berechnung für unterschiedliche Typen ist gleicher Name aber sinnvoll

- In C++ gehört Typ der Parameter zur Signatur

  ⇒ Compiler kann Funktionen mit gleichem Namen unterscheiden, wenn Typ und/oder Anzahl der Parameter verschieden ist

  ⇒ Unterschiedliche Signatur!

```
int Abs(int a)
{
    if (a<0)
        return -a;
    else
        return a;
}
```

```
double Abs(double a)
{
    if (a<0)
        return -a;
    else
        return a;
}
```

Funktionstyp und Überladen

- Was ist mit zwei Funktionen, die sich nur im Typ des Rückgabewertes unterscheiden?

```
int square(int a)
{
    return a*a;
}
```

```
double square(int a)
{
    return a*a;
}
```

⇒ Das ist nicht möglich!

- Keine eindeutige Zuordnung beim Funktionsaufruf möglich
- Typ der Funktion (Rückgabetyp) gehört nicht zur Signatur

```
int min(double a, double b)
{
    if (a<=b)
        return 1;
    else
        return 2;
}
```

```
double min(double a, double b)
{
    if (a<=b)
        return a;
    else
        return b;
}
```

Typkonvertierung und Überladen

```
int Abs(int a)
{
    if (a<0)
        return -a;
    else
        return a;
}
```

```
double Abs(double a)
{
    if (a<0)
        return -a;
    else
        return a;
}
```

```
long int li = 3L;
cout << Abs(li);
```

```
error:  call of overloaded 'Abs(long int)' is ambiguous
error:  candidates are:  int Abs(int)
error:                   double Abs(double)
```

- Compiler entscheidet, welche die richtige Funktion ist

  ⇒ Problem bei Typkonvertierung: keine eindeutige Entscheidung möglich, wenn mehrere Optionen bestehen

Flexibilität und Wiederverwendbarkeit

- Gleiche oder ähnliche Berechnungen/Anweisungen in Funktionen abbilden
- 1. Möglichkeit: Überladen von Funktionen für unterschiedliche Typen
- 2. Möglichkeit: Steuerung der Anweisungen in der Funktion durch Parameter

- Durch Parameter ist Funktionalität flexibel einsetzbar/wiederverwendbar
- Beispiel:

```
void printLine(int length, char c)
{
    for (int i=0; i<length; i++)
        cout << c;
    cout << endl;
}

int main()
{
    printLine(80,'-'); // Linie der Laenge 80 mit '-'
    printLine(20,'='); // Linie der Laenge 20 mit '='
    printLine(40,'X'); // Linie der Laenge 40 mit 'X'
}
```

Default-Parameter

- Wunsch: nur einen Teil der Parameter angeben, ansonsten "das Übliche"
- Im Beispiel: nur die Länge der Linie angeben, kein besonderes Zeichen (beliebiges Zeichen, Standard-Zeichen): `printLine(30);`
- Für alle Parameter "das Übliche": `printLine();`
  ⇒ Standardwerte (default-Werte) müssen vorher festgelegt werden, und zwar für jeden optionalen Parameter

```
void printLine(int length=80, char c='-')
{
    for (int i=0; i<length; i++)
        cout << c;
    cout << endl;
}

int main()
{
    printLine(); // Linie der Laenge 80 mit '-'
    printLine(20); // Linie der Laenge 20 mit '-'
    printLine(40,'='); // Linie der Laenge 40 mit '='
}
```

Verwendung von default-Parametern

- Weglassen von Parametern beim Funktionsaufruf nur von rechts nach links möglich
- Im Beispiel:
  - Nur Weglassen des Zeichens möglich (oder Weglassen aller Parameter)
  - Standardlänge aber spezifiziertes Zeichen nicht möglich: `printLine('=');`
- Nur einen Teil der Parameter mit default-Werten geht auch:
  `void printLine(int length, char c='-') ...`
- Dann muss die Länge immer angegeben werden, Fehlermeldung für den Aufruf `printLine();`
- In C++ gibt es keine "named arguments"
- Folgende Aufrufe funktionieren nicht:
  - `printLine(length=20,c='X');`
  - `printLine(c='X');`

Default-Parameter durch Überladen

- Anstelle von:

```
void printLine(int length=80, char c='-')
{
    for (int i=0; i<length; i++)
        cout << c;
    cout << endl;
}
```

- die Funktion printLine überladen:

```
void printLine(int length, char c)
{
    for (int i=0; i<length; i++)
        cout << c;
    cout << endl;
}
void printLine(int length)
{
    printLine(length,'-');
}
void printLine()
{
    printLine(80); // oder: printLine(80,'-');
}
```

Überladen ermöglicht Verwendung der Parameter in unterschiedlicher Reihenfolge

```
void printLine(int length, char c)
{
    for (int i=0; i<length; i++)
        cout << c;
    cout << endl;
}
```

```
void printLine(int length)
{
    printLine(length,'-');
}
```

```
void printLine(char c)
{
    printLine(80,c);
}
```

```
void printLine()
{
    printLine(80);
}
```

```
void printLine(char c, int length)
{
    printLine(length,c);
}
```

Inhalt

Funktionen: Deklaration und Definition

- Eine Funktion muss deklariert sein, bevor sie verwendet wird
- Deklaration teilt Compiler die Existenz einer Funktion mit
- Definition der Funktion kann auch später im Programm erfolgen

```
void printLine(int length, char c); // Deklaration
// Funktionskopf + Semikolon

int main()
{
    printLine(80,'-'); // Linie der Laenge 80 mit '-'
    printLine(20,'='); // Linie der Laenge 20 mit '='
    printLine(40,'X'); // Linie der Laenge 40 mit 'X'
}

void printLine(int length, char c) // Definition
{
    for (int i=0; i<length; i++)
        cout << c;
    cout << endl;
}
```

Default-Werte in der Deklaration ausreichend

```
void printLine(int length=80, char c='-'); // Deklaration
// Funktionskopf + Semikolon

int main()
{
    printLine(80,'-'); // Linie der Laenge 80 mit '-'
    printLine(20,'='); // Linie der Laenge 20 mit '='
    printLine(40,'X'); // Linie der Laenge 40 mit 'X'
}

void printLine(int length, char c) // Definition
{
    for (int i=0; i<length; i++)
        cout << c;
    cout << endl;
}
```

Funktionen richtig einsetzen

- Generelle Empfehlung: ein Problem in viele kleine Funktionen zerlegen
- Das hat natürlich Grenzen, aber am Anfang neigt man dazu zu wenig Funktionen zu schreiben

- Fein aufgegliederte Funktionen sind überschaubar, gut zu schreiben und gut zu testen
- Funktionen mit komplexen Aufgaben bleiben überschaubar, wenn man sie in kleine Teilfunktionen zerlegt
- Viele kleine Funktionen erhöhen Wahrscheinlichkeit der Wiederverwendbarkeit

- Jede Funktion löst klar abgrenzbare Aufgabe:
  - Berechnung und Ausgabe nicht in einer Funktion
  - Berechnung in einer Funktion, Ausgabe in weiterer Funktion

Funktionsnamen

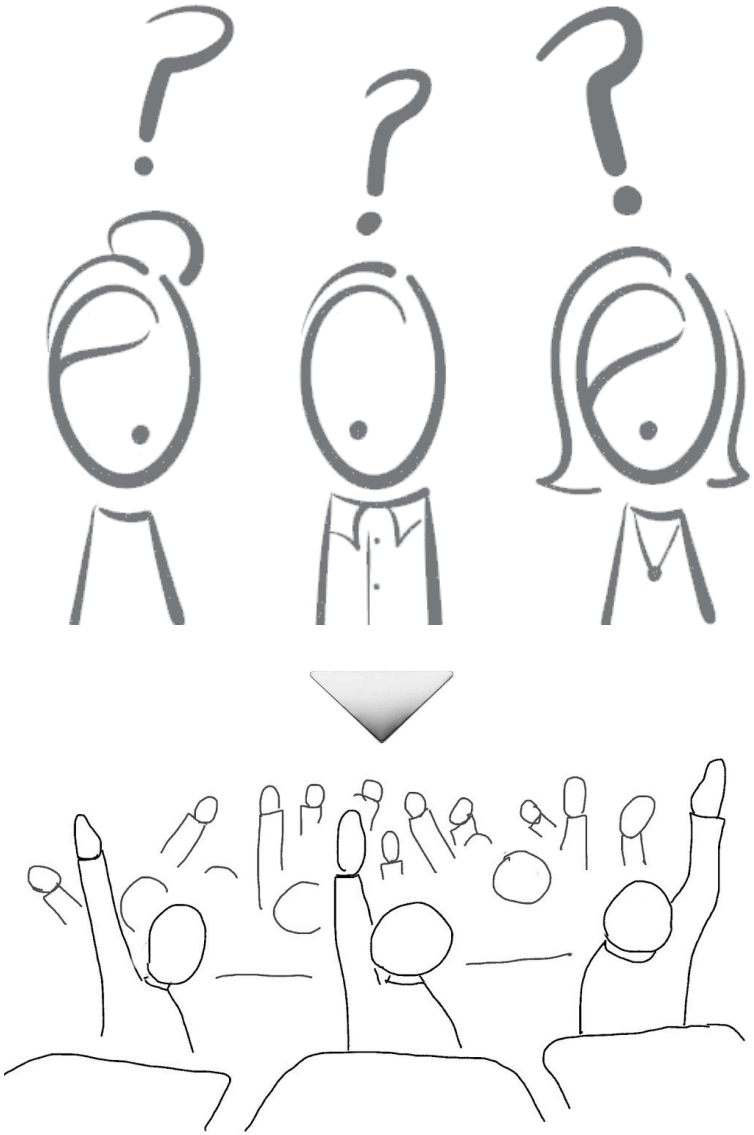
- Aussagekräftige Funktionsnamen
- Der Name sagt, was die Funktion tut
- Typisch: Verb + Substantiv

- 1. Möglichkeit (camel-case): drawCircle, addValue, setStart, getWidth, isEqual, calculateNumberOfComponents
- 2. Möglichkeit (mit Unterstrich): draw_circle, add_value, set_start, get_width, is_equal, calculate_number_of_components

- Auch möglich:
  - Variablennamen als camel-case (firstName, numberOfPixels)
  - Funktionsnamen mit Unterstrich

- Ausnahme: mathematische Funktionen, die einen Wert berechnen, können danach heißen, was sie berechnen, z.B. sin()

Gibt es Fragen? (Es gibt keine dummen Fragen!)



"Excuse me, is this the Society for Asking Stupid Questions?"