Funktionen

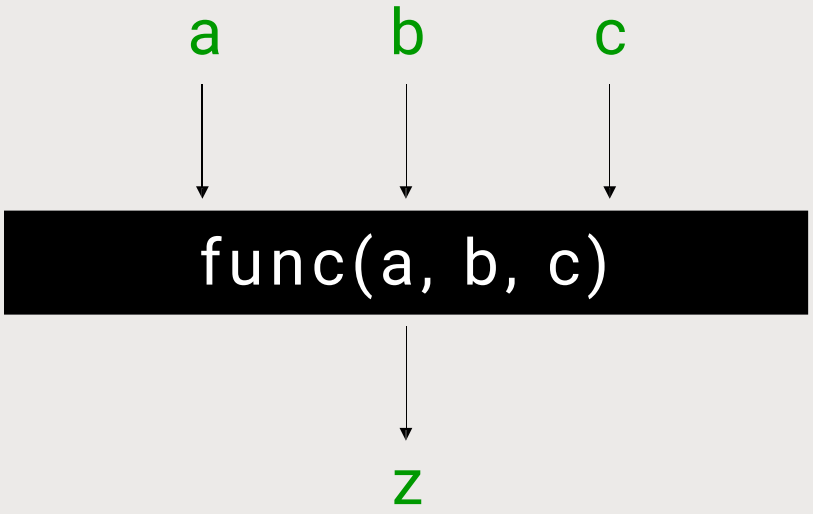
>> Bis jetzt wurden alle Programme, die wir in diesem Kurs geschrieben haben, innerhalb von `main()` geschrieben.

Das war bisher noch kein Problem, aber es könnte eines werden, wenn unsere Programme anfangen, unhandlich zu werden.

>> C und fast alle Sprachen, die seitdem entwickelt wurden, erlauben es uns, **Funktionen** zu schreiben, die manchmal auch als **Prozeduren**, **Methoden** oder **Unterprogramme** bezeichnet werden.

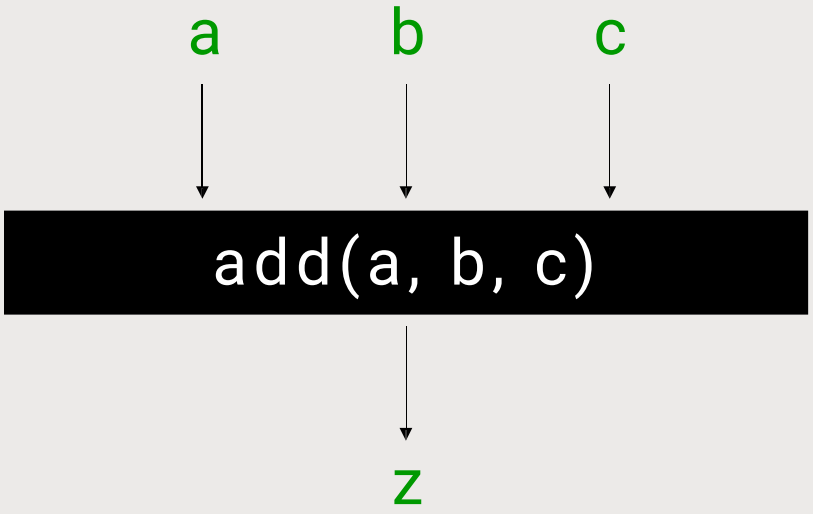
Was ist eine Funktion?

Eine Black-Box mit keinem oder mehreren Eingabewerten und einem Ausgabewert.



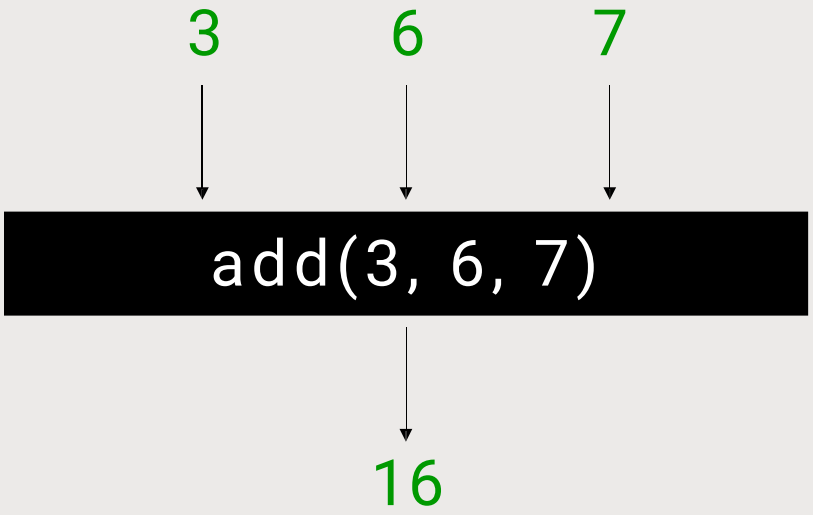
Was ist eine Funktion?

Eine Black-Box mit keinem oder mehreren Eingabewerten und einem Ausgabewert.



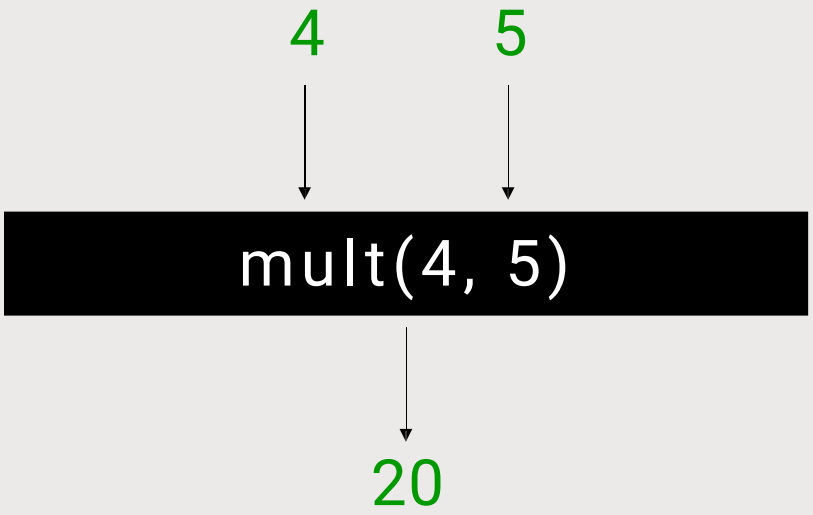
Was ist eine Funktion?

Eine Black-Box mit keinem oder mehreren Eingabewerten und einem Ausgabewert.



Was ist eine Funktion ?

Eine Black-Box mit keinem oder mehreren Eingabewerten und einem Ausgabewert.



Warum sagen wir Black-Box?

Wenn wir die Funktionen nicht selbst schreiben, müssen wir die zugrunde liegende Implementierung nicht kennen.

```
mult(a, b):
  output a * b
```

Warum sagen wir Black-Box?

Wenn wir die Funktionen nicht selbst schreiben, müssen wir die zugrunde liegende Implementierung nicht kennen.

```
mult(a, b):
  set counter to 0
  repeat b times
     add a to counter
  output counter
```

Warum sagen wir Black-Box?

Wenn wir die Funktionen nicht selbst schreiben, müssen wir die zugrunde liegende Implementierung nicht kennen.

Funktionen haben idealerweise klare und **offensichtliche Namen** und sind gut dokumentiert.

Ihr Verhalten ist **idealerweise vorher-sehbar** basierend auf dem Namen.

Es ist so ähnlich wie bei einem Vertrag, mit dem **bestimmte Eigenschaften zugesichert** werden: Wenn man eine Funktion schreibt, die andere (oder man selbst) verwenden, geht man davon aus, dass die Funktion sich so verhält, wie man es erwarten würde.

Funktionen helfen bei der Abstraktion.

Organisation

Funktionen helfen, ein kompliziertes Problem in überschaubare Teilaufgaben zu zerlegen.

Vereinfachung

Kleinere Komponenten sind in der Regel einfacher zu entwerfen, zu implementieren und zu debuggen.

Wiederverwendbarkeit

Funktionen können recycelt werden; man muss sie nur einmal schreiben, kann sie aber so oft wie nötig verwenden.

Funktions-deklarationen

Der erste Schritt zur Erstellung einer Funktion ist ihre Deklaration (auch **Prototyp** genannt).

Funktions-prototypen

Dadurch bekommt der Compiler einen Hinweis, dass eine Funktion irgendwo im Code vorkommen wird.

Funktionsdeklarationen sollten immer am Anfang von Dateien stehen, d.h. vor dem Beginn von `main()`.

Es gibt eine Standardform für
jede Funktionsdeklaration:

```
rückgabe-typ name(argumentliste);
```

Der **Rückgabetyp** ist die Art der Variable, die die Funktion ausgeben wird.

Der **Name** gibt an, wie Sie Ihre Funktion nennen möchten.

Die **Argumentliste** ist die durch Kommas getrennte Liste von Eingabewerten für Ihre Funktion, jede mit ihrem Typ und ihrem Namen.

Am Ende steht ein Strichpunkt.

Beispiel 1 Eine Funktion zum Addieren zweier ganzer Zahlen:

```
int add_two_integers(int a, int b);
```

Die **Summe** von zwei Ganzzahlen wird ebenfalls eine Ganzzahl sein, also ist der Rückgabetyp `int`.

Der **Name** deutet an, dass die Funktion zwei int-Werte addiert.

Die **Argumentliste** enthält zwei durch Kommas getrennte Variablen `a` und `b`, die jeweils den Datentyp `int` haben.

Strichpunkt nicht vergessen!

Beispiel 2

Eine Funktion zum Multiplizieren zweier Fließkommazahlen:

```
float mult_two_reals(float a, float b);
```

Das Produkt zweier Fließkommazahlen ist ebenfalls eine Fließkommazahl.

Geben wir dem Ganzen einen passenden Namen.

Auch hier sind die Namen der Eingabewerte zum Verständnis nicht so wichtig, also können wir kurze und einfache Namen (a, b) verwenden.

Alternative mit höherer Genauigkeit

```
double mult_two_reals(double a, double b);
```

Funktions-definition

Der zweite Schritt zur Erstellung einer Funktion ist ihre **Definition**. Sie beschreibt das Verhalten beim Funktionsaufruf. Die Definition beginnt wie die Deklaration – jetzt aber **ohne Strich-punkt**! Danach folgt die Implementation:

```
float mult_two_reals(float x, float y)
{
    float product = x * y;
    return product;
}
```

oder kürzer

```
float mult_two_reals(float x, float y)
{
    return x * y;
}
```

Übung zu Funktions-definitionen

Definieren Sie die Funktion

```
int add_two_ints(int x, int y);
```

```
int add_two_ints(int x, int y)
{
    int sum = x + y;
    return sum;
}
```

oder kürzer

```
int add_two_ints(int x, int y)
{
    return x * y;
}
```

Oder umständlicher:

```c
int add_two_ints(int a, int b)
{
    int sum = 0;
    if (a > 0)
        for (int i = 0; i < a; sum++, i++)
            ;
    else
        for (int i = a; i < 0; sum--, i++)
            ;
    if (b > 0)
        for (int i = 0; i < b; sum++, i++)
            ;
    else
        for (int i = b; i < 0; sum--, i++)
            ;
    return sum;
}
```

>> Merke: Nur weil es funktioniert,
heißt das nicht, dass es gut ist!

KISS-Prinzip:
Keep it simple and stupid.

Funktions-aufruf

Um eine Funktion zu verwenden, rufen wir sie in unserem Code auf.

Dabei müssen wir der Funktion passende Argumente übergeben und den Rückgabewert einer Variable des richtigen Typs zuweisen.

Schauen wir uns ein Beispiel an.

```c
#include <cs50.h>
#include <stdio.h>


int add_two_ints(int a, int b);


int main(void)
{
    int x = get_int("Geben Sie eine Zahl ein: ");
    int y = get_int("Geben Sie noch eine Zahl ein: ");


    int z = add_two_ints(x, y);
    printf("Die Summe von %i und %i ist %i\n", x, y, z);
}
```

Weitere Hinweise

Funktionen können manchmal keine Eingaben haben. In diesem Fall deklarieren wir die Funktion mit einer *void*-Argumentenliste:

```
int main(void)
```

Funktionen geben manchmal keinen Wert zurück. In diesem Fall deklarieren wir die Funktion mit einem *void*-Rückgabetyp:

```
void print_hello(void)
{
    printf("Hallo, Welt!\n");
}
```

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Übung

Deklarieren und schreiben Sie eine Funktion `valid_triangle`:

Die Funktion nimmt drei reelle Zahlen als Argumente (Seitenlängen eines Dreiecks) entgegen und gibt *true* oder *false* zurück. Sie prüft, ob die Längen ein gültiges Dreieck bilden können, d.h.:

1. Alle Seiten müssen eine positive Länge haben.
2. Die Summe zweier Seiten muss größer sein als die Länge der dritten Seite.

Beginnen Sie mit dem Prototyp.

```
bool valid_triangle(float x, float y, float z)
{
    // Alle Seiten müssen positiv sein.
    if (x <= 0 || y <= 0 || z <= 0)
    {
        return false;
    }
    // Die Summe zweier Seiten muss größer sein als die dritte.
    if ((x + y <= z) || (x + z <= y) || (y + z <= x))
    {
        return false;
    }


    // Beide Tests bestanden, also gültiges Dreieck.
    return true;
}
```

F1: Warum muss eine Funktion oberhalb der *main*-Funktion deklariert werden?

F2: Könnte man eine Funktion auch einfach direkt oberhalb der *main*-Funktion definieren? Wieso ist dieses Vorgehen nicht üblich?

F3: Was passiert, wenn eine Funktion einen *int* als Argument erwartet, aber ein *float* übergeben wird?

Funktionen: Black Box mit optionalen Eingabewerten und einem Rückgabewert

Definition:
Implementation des Funktionsverhaltens

Beginnt wie Prototyp,
aber ohne Strichpunkt.

Vorteile:

Organisation, Vereinfachung, Wiederverwendbarkeit

ermöglichen Abstraktion

void:
keine Rückgabe oder
keine Parameter

Deklaration/Prototyp:
Rückgabetyp, Name, Argumentliste

steht i.d.R. vor main()

KISS:
Keep It Simple and Stupid
Einfache Implementationen sind zu bevorzugen.