Einführung in die Programmierung

Vorlesungsteil 5
Funktionen selbst programmieren und Module bauen

PD Dr. Thomas Hinze

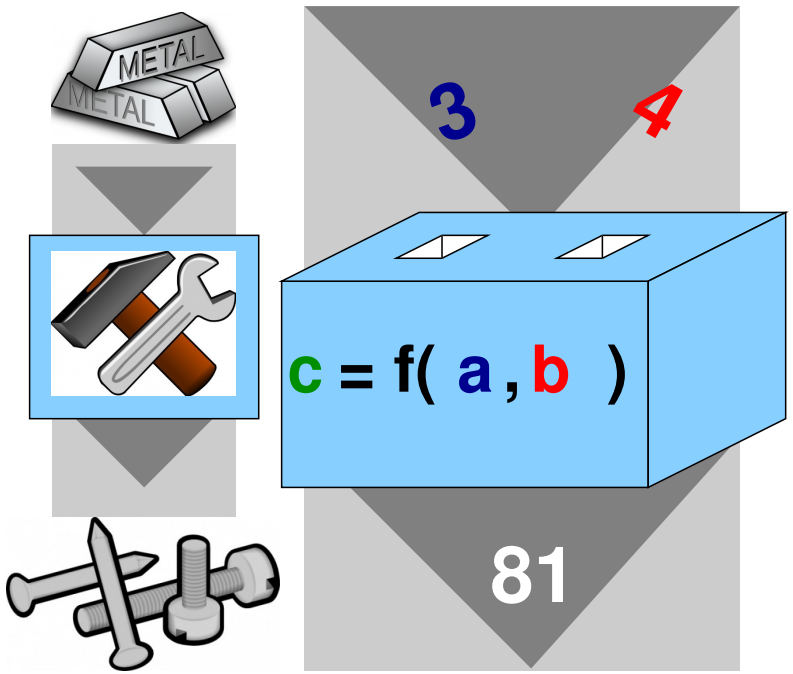
Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Funktion als Werkzeug, das auf Daten einwirkt

Der Begriff **Funktion** stammt aus dem Lateinischen und bedeutet *„Tätigkeit“* oder *„Verrichtung“* im Sinne einer zielgerichteten Anwendung eines **Werkzeugs** auf **Werkstücke**. Mathematik und Informatik greifen den Begriff Funktion auf, wobei die Werkstücke dann Datenwerte sind.



Funktionen als zentrale Bausteine der C-Programmierung

"Ein C–Programm ist eine Sammlung von Funktionen, die sich gegenseitig aufrufen, beginnend mit der main–Funktion."

Grundlegende Erkenntnis des C–Programmierers

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Wofür sind Funktionen sinnvoll?

Beispiel: Eine Funktion zur Berechnung von Nullstellen quadratischer Polynome

$0 = x^2 + px + q$ Meine Parameter: $p = -6.1561, q = 8.03636$

Wofür sind Funktionen sinnvoll?

Beispiel: Eine Funktion zur Berechnung von Nullstellen quadratischer Polynome

$0 = x^2 + px + q$ Meine Parameter: $p = 3.9876, q = -15.353$

Wofür sind Funktionen sinnvoll?

Beispiel: Eine Funktion zur Berechnung von Nullstellen quadratischer Polynome

$0 = x^2 + px + q$ Meine Parameter: $p = -0.701, q = -2.2222$

Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.

Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.
- Das kann auch an unterschiedlichen Stellen innerhalb ein und desselben Programms der Fall sein, denken Sie zum Beispiel an einen Sortieralgorithmus, der in einer Tabellenkalkulation häufig benötigt wird.

Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.
- Das kann auch an unterschiedlichen Stellen innerhalb ein und desselben Programms der Fall sein, denken Sie zum Beispiel an einen Sortieralgorithmus, der in einer Tabellenkalkulation häufig benötigt wird.
- Es wäre als Programmierer unklug, dann jedes Mal „das Fahrrad neu zu erfinden“ und die Berechnungsvorschrift jedes Mal erneut gesondert in den Quelltext zu schreiben.

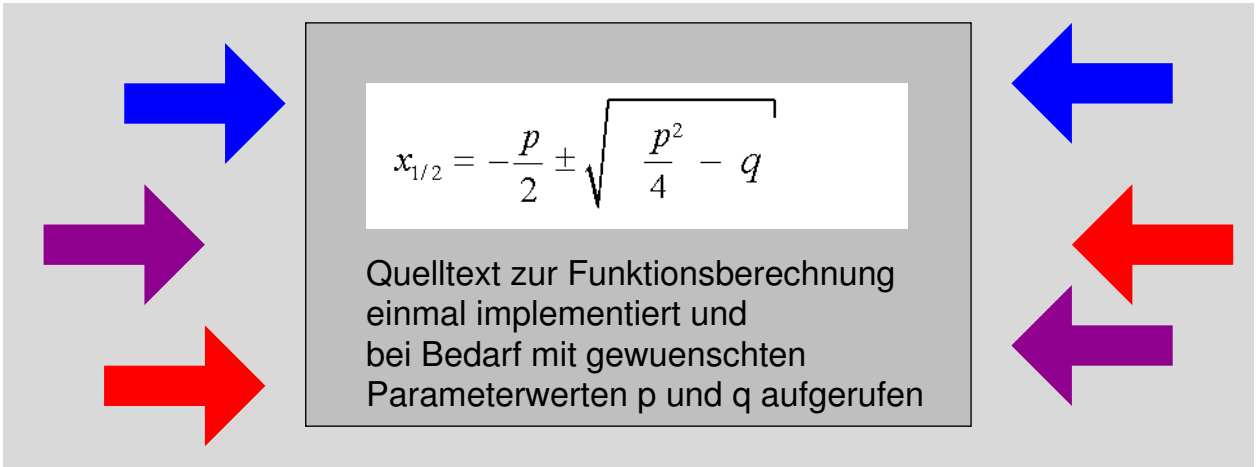
Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.
- Das kann auch an unterschiedlichen Stellen innerhalb ein und desselben Programms der Fall sein, denken Sie zum Beispiel an einen Sortieralgorithmus, der in einer Tabellenkalkulation häufig benötigt wird.
- Es wäre als Programmierer unklug, dann jedes Mal „das Fahrrad neu zu erfinden“ und die Berechnungsvorschrift jedes Mal erneut gesondert in den Quelltext zu schreiben.

$\Longrightarrow$ *Funktionen* ermöglichen es, eine Berechnungsvorschrift oder einen Algorithmus *einmal* zu *implementieren* und diese Implementierung *immer bei Bedarf* wieder *aufzurufen*.

Vorteile von Funktionen in der Programmierung

Wiederverwendbarkeit von Quelltext

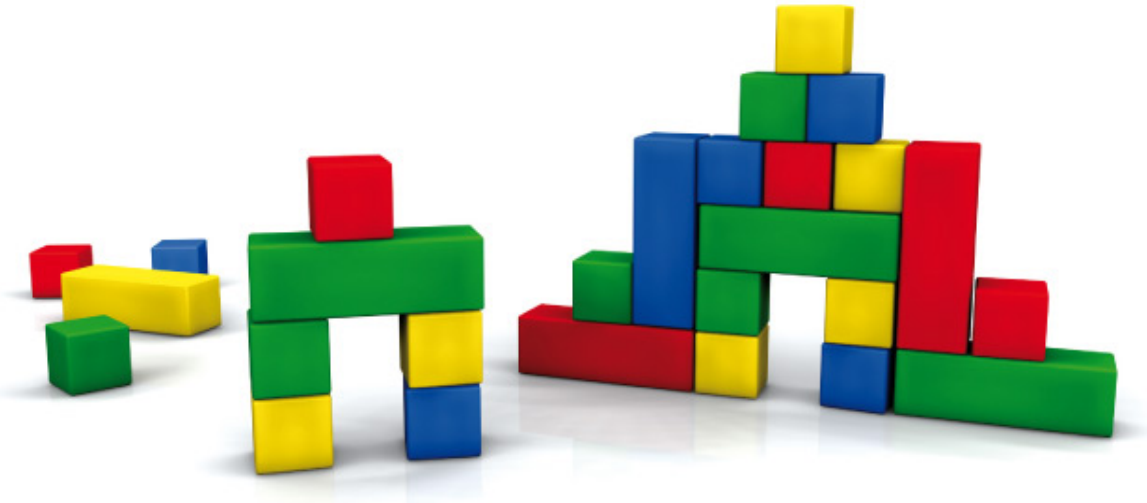


- führt zu deutlich kürzeren Quelltexten mit weniger Redundanz
- führt zu weniger Fehlern im Quelltext
- führt zu effizienterem Programmieren
  (mehr Features pro Zeiteinheit implementierbar)

$\Longrightarrow$ Funktionen als wiederverwendbare *Werkzeuge*

Vorteile von Funktionen in der Programmierung

Leichtere Wartbarkeit von Software



- Änderungen nur an einer Stelle im Quelltext vorzunehmen
- Implementierung einer Funktion unabhängig vom restlichen Quelltext austauschbar, Gesamtquelltext überschaubarer
- Funktion leicht ersetzbar durch schnellere, genauere oder einfach nur fehlerbereinigte Version

$\Longrightarrow$ Funktionen als anwendungsfertige *Bausteine*

Vorbild: Funktionen in der Mathematik

$$f_1 : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

$$f_1(p, q) = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$$

- Funktion besitzt einen klar zuordenbaren ***Namen***, z.B. f_1
- Funktion besitzt ***Argumente*** (Parameter), die bei Aufruf mit konkreten Werten aus Definitionsbereich belegt werden, z.B. p und q

Vorbild: Funktionen in der Mathematik

Aufruf:

$y = f_1(-6.1561, 8.03636)$

$$f_1 : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

$$f_1(p, q) = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$$

- Funktion besitzt einen klar zuordenbaren ***Namen***, z.B. f_1
- Funktion besitzt ***Argumente*** (Parameter), die bei Aufruf mit konkreten Werten aus Definitionsbereich belegt werden, z.B. p und q
- In der Reihenfolge der Argumente von links nach rechts werden die Werte zugeordnet, also z.B. $p = -6.1561$ und $q = 8.03636$
- Funktion berechnet den Funktionswert und gibt ihn als Berechnungsergebnis zurück

Vorbild: Funktionen in der Mathematik

Aufruf:
$y = f_1(-6.1561, 8.03636)$

Aufruf:
$z = f_1(\frac{\pi}{2} + 1, 2 \cdot 0.15)$

$$f_1 : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

$$f_1(p, q) = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$$

- Funktion besitzt einen klar zuordenbaren ***Namen***, z.B. f_1
- Funktion besitzt ***Argumente*** (Parameter), die bei Aufruf mit konkreten Werten aus Definitionsbereich belegt werden, z.B. p und q
- In der Reihenfolge der Argumente von links nach rechts werden die Werte zugeordnet, also z.B. $p = -6.1561$ und $q = 8.03636$
- Funktion berechnet den Funktionswert und gibt ihn als Berechnungsergebnis zurück
- Funktion kann erneut (beliebig oft) mit weiteren Werten aufgerufen werden

Vorbild: Funktionen in der Mathematik

$$g : \mathbb{R} \longrightarrow \mathbb{R} \qquad f : \mathbb{R} \longrightarrow \mathbb{R}$$
$$g(x) = \sin(x) \qquad f(x) = x^2$$

Aufruf: $z = f(g(\frac{\pi}{4})) = f(\sin(\frac{\pi}{4})) = \left(\sin(\frac{\pi}{4})\right)^2$

- Funktionsaufrufe dürfen ineinander verschachtelt werden (beliebig, aber endlich tief)

Vorbild: Funktionen in der Mathematik

$$g : \mathbb{R} \longrightarrow \mathbb{R} \qquad f : \mathbb{R} \longrightarrow \mathbb{R}$$
$$g(x) = \sin(x) \qquad f(x) = x^2$$

Aufruf: $z = f(g(\frac{\pi}{4})) = f(\sin(\frac{\pi}{4})) = \left(\sin(\frac{\pi}{4})\right)^2$

- Funktionsaufrufe dürfen ineinander verschachtelt werden (beliebig, aber endlich tief)
- Aus der Verschachtelung in Kombination mit Priorisierung der genutzten Operationen und Abarbeitung von links nach rechts bei gleichrangigen Operationen resultiert klare Reihenfolge, in der die Berechnungsschritte ausgeführt werden

Vorbild: Funktionen in der Mathematik

$$g : \mathbb{R} \longrightarrow \mathbb{R} \qquad f : \mathbb{R} \longrightarrow \mathbb{R}$$
$$g(x) = \sin(x) \qquad f(x) = x^2$$

Aufruf: $z = f(g(\frac{\pi}{4})) = f(\sin(\frac{\pi}{4})) = \left(\sin(\frac{\pi}{4})\right)^2$

- Funktionsaufrufe dürfen ineinander verschachtelt werden (beliebig, aber endlich tief)
- Aus der Verschachtelung in Kombination mit Priorisierung der genutzten Operationen und Abarbeitung von links nach rechts bei gleichrangigen Operationen resultiert klare Reihenfolge, in der die Berechnungsschritte ausgeführt werden
- Bei Funktionsaufruf spielt die Benennung der Argumente in der Funktionsdefinition keine Rolle, es ist also egal, ob die Funktion g definiert ist als $g(x) = \sin(x)$ oder $g(w) = \sin(w)$ oder z.B. $g(y) = \sin(y)$

Funktionsdefinition in C: Syntaktische Struktur

Jede Funktionsdefinition in C wird wie folgt im Quelltext notiert:

```
<Rückgabetyp> <Name>(<getypte Argumentliste>)
{
   <Funktionsrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Funktionsnamen

Funktionsdefinition in C: Syntaktische Struktur

Jede Funktionsdefinition in C wird wie folgt im Quelltext notiert:

```
<Rückgabetyp> <Name>(<getypte Argumentliste>)
{
   <Funktionsrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Funktionsnamen
- Die `<getypte Argumentliste>` enthält durch Kommas getrennt selbstgewählte Namen für die einzelnen Argumente, vor jedes Argument wird noch der entsprechende Typ gesetzt

Funktionsdefinition in C: Syntaktische Struktur

Jede Funktionsdefinition in C wird wie folgt im Quelltext notiert:

```
<Rückgabetyp> <Name>(<getypte Argumentliste>)
{
   <Funktionsrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Funktionsnamen
- Die `<getypte Argumentliste>` enthält durch Kommas getrennt selbstgewählte Namen für die einzelnen Argumente, vor jedes Argument wird noch der entsprechende Typ gesetzt
- `{<Funktionsrumpf>}` entspricht einem Block. Darin können lokale Variablen oder Konstanten vereinbart werden, die nur innerhalb der Funktion benötigt werden und deren Speicherplatz nach Abarbeitung des Funktionsaufrufs wieder freigegeben wird

Funktionsdefinition in C: Syntaktische Struktur

Jede Funktionsdefinition in C wird wie folgt im Quelltext notiert:

```
<Rückgabetyp> <Name>(<getypte Argumentliste>)
{
   <Funktionsrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Funktionsnamen
- Die `<getypte Argumentliste>` enthält durch Kommas getrennt selbstgewählte Namen für die einzelnen Argumente, vor jedes Argument wird noch der entsprechende Typ gesetzt
- `{<Funktionsrumpf>}` entspricht einem Block. Darin können lokale Variablen oder Konstanten vereinbart werden, die nur innerhalb der Funktion benötigt werden und deren Speicherplatz nach Abarbeitung des Funktionsaufrufs wieder freigegeben wird
- Jeder Abarbeitungspfad im Funktionsrumpf muss mit einer `return`-Anweisung enden. Dahinter wird der zurückgegebene Funktionswert vom Rückgabetyp geschrieben.

Funktion zur Berechnung des Rechteckumfangs

rechteckumfang.c

```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}


int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Merke

- Ohne weitere Vorkehrungen im Quelltext definiert man in C eigene Funktionen *oberhalb* der `main`-Funktion, unmittelbar hinter den Präprozessorbefehlen

Merke

- Ohne weitere Vorkehrungen im Quelltext definiert man in C eigene Funktionen *oberhalb* der `main`-Funktion, unmittelbar hinter den Präprozessorbefehlen
- Bei mehreren selbstdefinierten Funktionen müssen sich deren Namen in C immer unterscheiden, auch dann, wenn sie typverschiedene Argumentlisten haben. Angenommen, Sie schreiben Funktionen zur Kubikwurzelberechnung. Eine Funktion tut dies im Bereich ganzer Zahlen, z.B. `int cubicroot(int x)`, die andere für Gleitkommazahlen. Diese darf dann *nicht* `double cubicroot(double x)` heißen, sondern muss einen anderen Namen bekommen.

Merke

- Ohne weitere Vorkehrungen im Quelltext definiert man in C eigene Funktionen *oberhalb* der `main`-Funktion, unmittelbar hinter den Präprozessorbefehlen
- Bei mehreren selbstdefinierten Funktionen müssen sich deren Namen in C immer unterscheiden, auch dann, wenn sie typverschiedene Argumentlisten haben. Angenommen, Sie schreiben Funktionen zur Kubikwurzelberechnung. Eine Funktion tut dies im Bereich ganzer Zahlen, z.B. `int cubicroot(int x)`, die andere für Gleitkommazahlen. Diese darf dann *nicht* `double cubicroot(double x)` heißen, sondern muss einen anderen Namen bekommen.
- Selbstdefinierte Funktionen dürfen sich auch *gegenseitig* aufrufen, z.B. dürfte eine Funktion `quaderkantenumfang` ihrerseits die Funktion `rechteckumfang` aufrufen. Ohne weitere Vorkehrungen im Quelltext muss auf die Reihenfolge der Funktionsdefinitionen geachtet werden, so dass *Neues stets auf Bekanntes zurückgeführt* wird.

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
float f(float x) {...}
float g(float x) {...}
float h(float x) {...}
```

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
float f(float x) {...}
float g(float x) {...}
float h(float x) {...}
```

Dann ist der aufrufende Quelltext

```
float y = ...;
float a = f(y);
float b = g(a);
float c = h(b);
```

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
float f(float x) {...}
float g(float x) {...}
float h(float x) {...}
```

Dann ist der aufrufende Quelltext

```
float y = ...;
float a = f(y);
float b = g(a);
float c = h(b);
```

semantisch äquivalent zu

```
float y = ...;
float c = h(g(f(y)));
```

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
float f(float x) {...}
float g(float x) {...}
float h(float x) {...}
```

Dann ist der aufrufende Quelltext

```
float y = ...;
float a = f(y);
float b = g(a);
float c = h(b);
```

semantisch äquivalent zu

```
float y = ...;
float c = h(g(f(y)));
```

Merke: Die „Transfervariablen“ `a` und `b` brauchen bei Aufrufschachtelung nicht angelegt zu werden.

Übergabe der Parameterwerte (call by value)

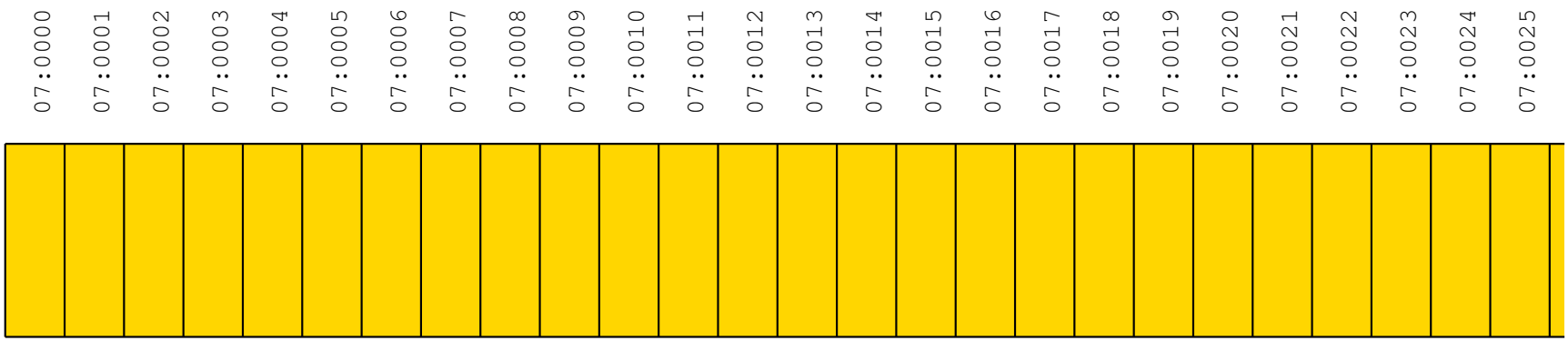
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Wie erfolgt Übergabe der Parameterwerte (Argumente) im Speicher?

Zu welchen Zeitpunkten ist wofür Speicherplatz reserviert?

Übergabe der Parameterwerte (call by value)

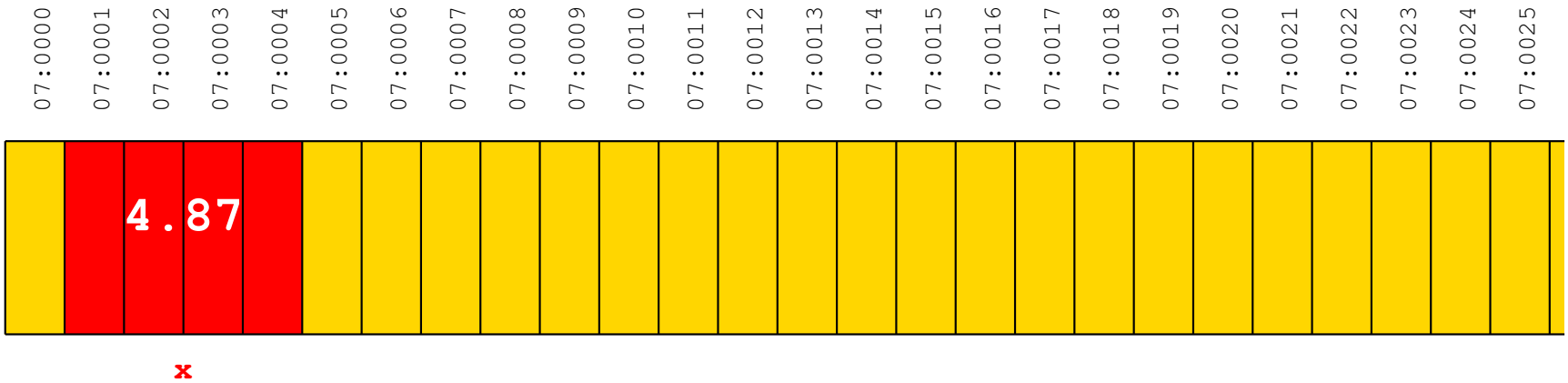
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Für die **`float`**-Variable **`x`** werden 4 Bytes (32 Bit) im Speicher reserviert und mit dem Bitmuster des Wertes **`4.87`** gemäß Kodierung IEEE754 belegt.

Übergabe der Parameterwerte (call by value)

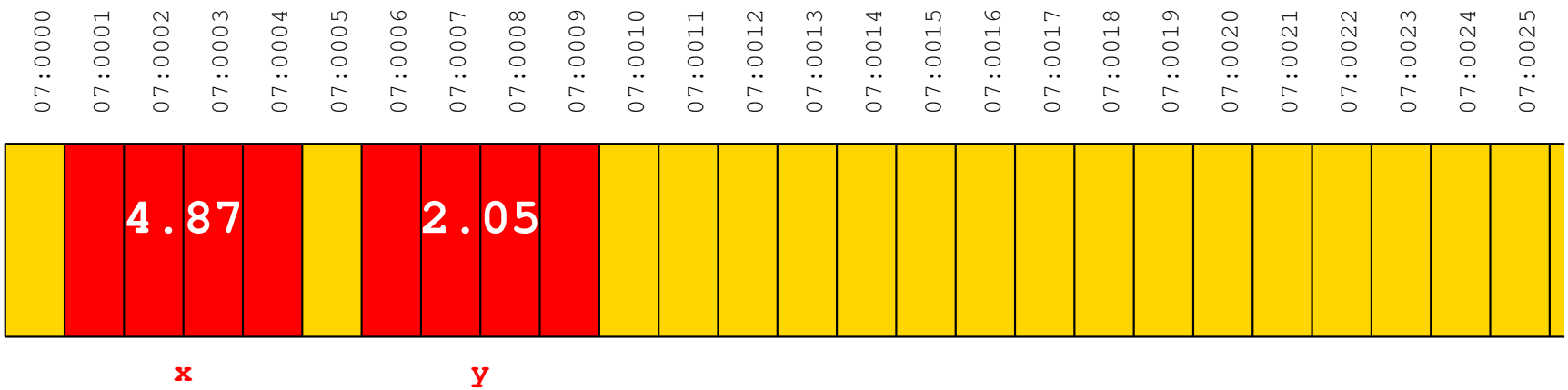
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Für die `float`-Variable `y` ebenfalls 4 Bytes (32 Bit) im Speicher reserviert und mit dem Bitmuster des Wertes `2.05` gemäß Kodierung IEEE754 belegt.

Übergabe der Parameterwerte (call by value)

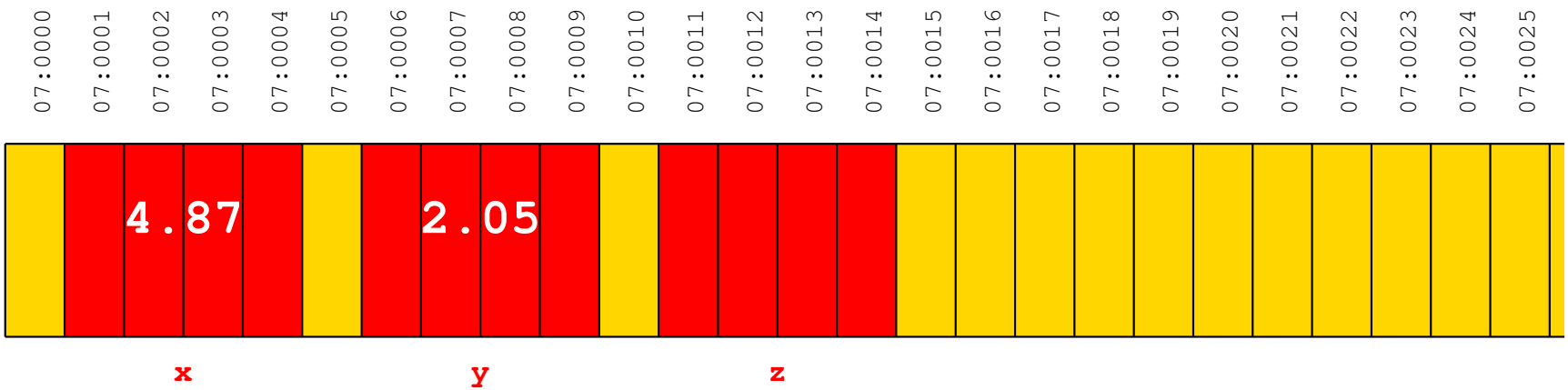
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Für die **`float`**-Variable **`z`** auch 4 Bytes (32 Bit) im Speicher reserviert. Das dort vorgefundene Bitmuster wird als Wert gemäß Kodierung IEEE754 interpretiert.

Übergabe der Parameterwerte (call by value)

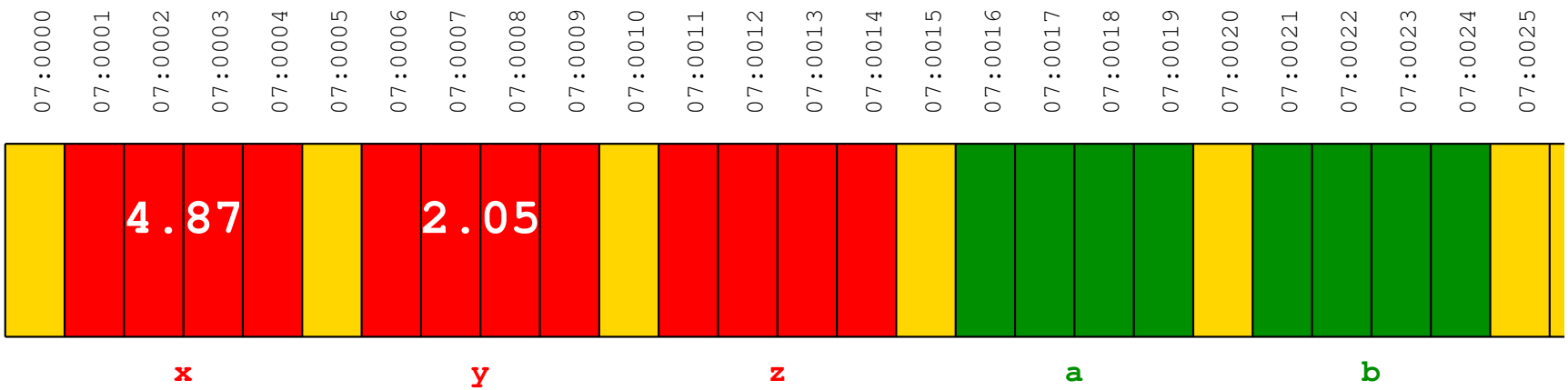
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Bei Funktionsaufruf für jedes Argument eine entsprechende Variable angelegt und dafür *neuer Speicherplatz* reserviert, hier also für die **`float`**-Variablen **`a`** und **`b`**.

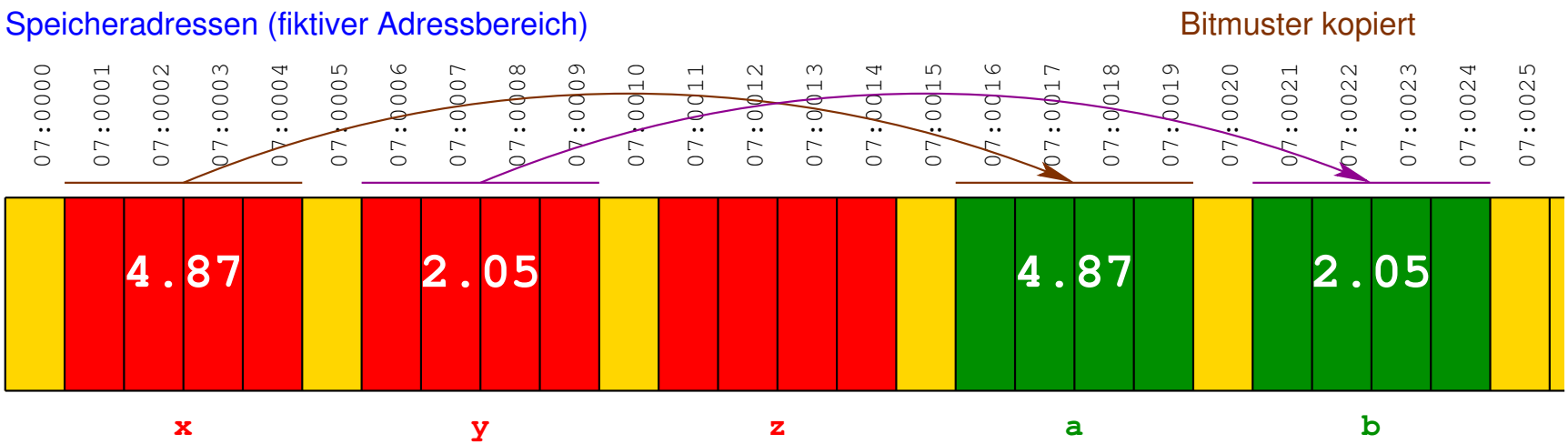
Übergabe der Parameterwerte (call by value)

```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```



Das Bitmuster des Speicherbereiches **x** wird in den Speicherbereich **a** *kopiert* und das Bitmuster des Speicherbereiches **y** in den Speicherbereich **b**.

Übergabe der Parameterwerte (call by value)

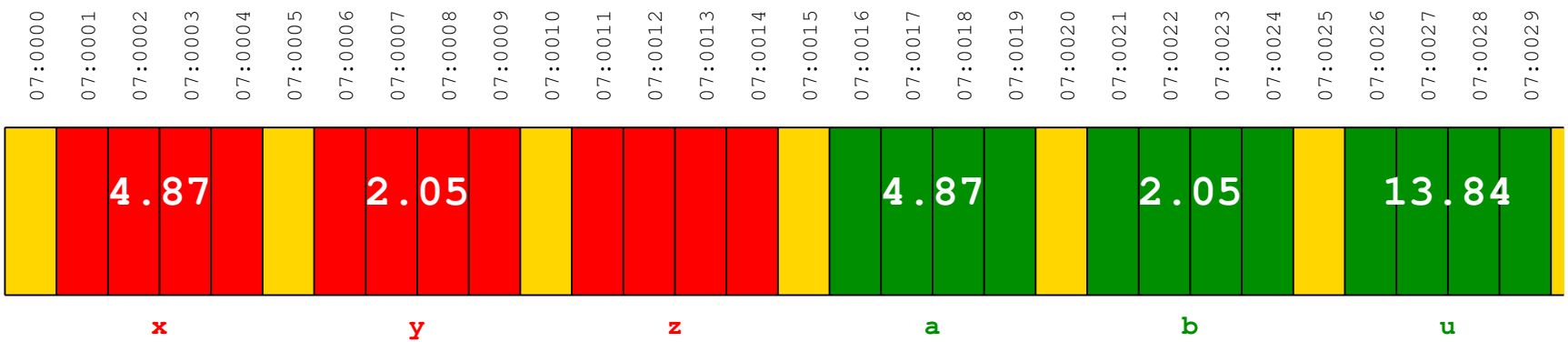
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Für die `float`-Variable `u` werden 4 Bytes (32 Bit) im Speicher reserviert und mit dem Bitmuster des `float`-Wertes `13.84` $(2 \cdot (4.87 + 2.05))$ belegt.

Übergabe der Parameterwerte (call by value)

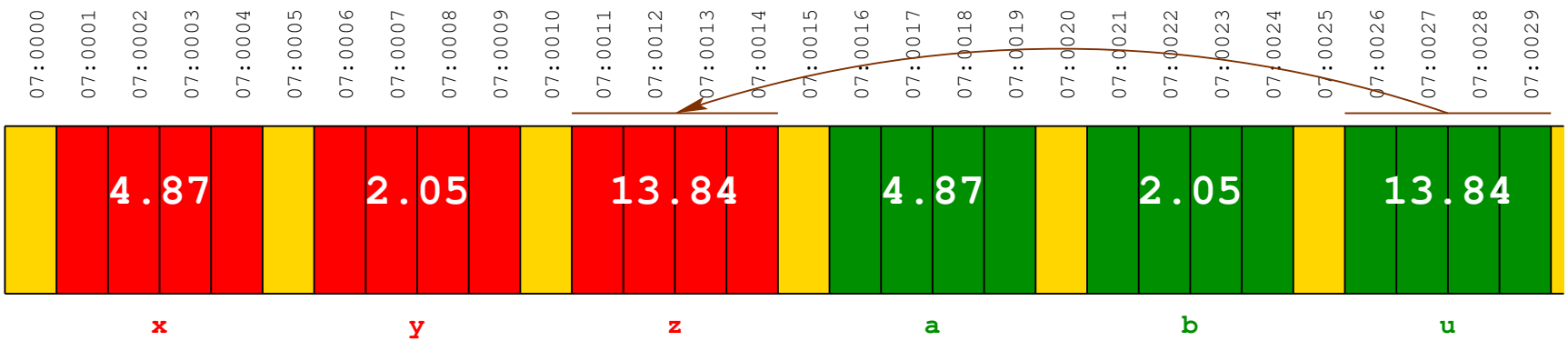
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Das Bitmuster des Speicherbereiches **u** wird in den Speicherbereich **z** *kopiert* zur Rückgabe des berechneten Funktionswertes.

Übergabe der Parameterwerte (call by value)

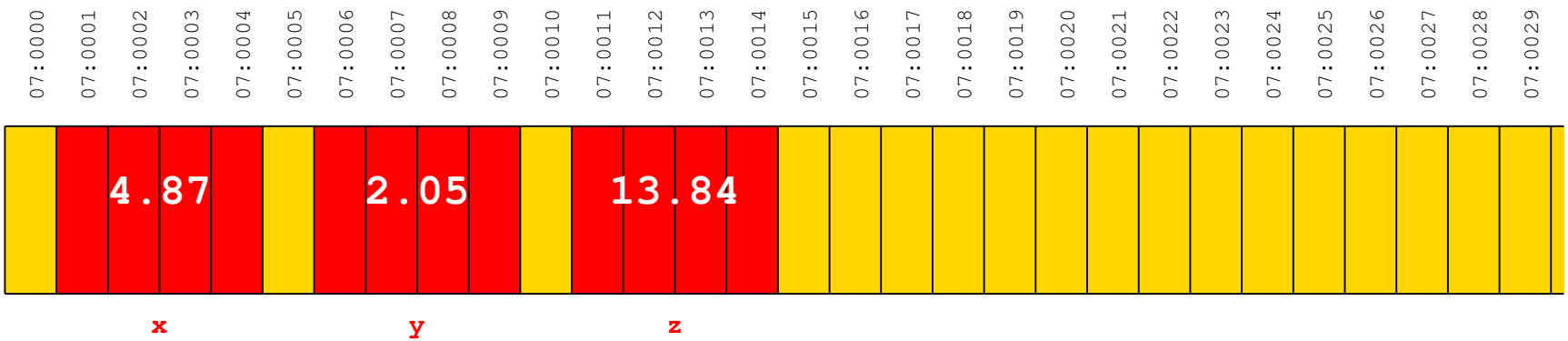
```c
#include <stdio.h>

float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Bei Erreichen des Blockendes der Funktion **`rechteckumfang`** werden alle funktionslokal ausgefassten Speicherbereiche wieder freigegeben, die Variablen **`a`**, **`b`** und **`u`** existieren nicht mehr.

Fragen

- Das Prinzip der Werteübergabe durch Kopieren (call by value) scheint leicht nutzbar zu sein, wenn es in jeder Funktion *nur einen Rückgabewert* als Funktionswert gibt.

Fragen

- Das Prinzip der Werteübergabe durch Kopieren (call by value) scheint leicht nutzbar zu sein, wenn es in jeder Funktion *nur einen Rückgabewert* als Funktionswert gibt.
- Es sind aber Funktionen denkbar, die *mehr als einen Wert zurückgeben.*

Fragen

- Das Prinzip der Werteübergabe durch Kopieren (call by value) scheint leicht nutzbar zu sein, wenn es in jeder Funktion *nur einen Rückgabewert* als Funktionswert gibt.
- Es sind aber Funktionen denkbar, die *mehr als einen Wert zurückgeben.*
- Stellen wir uns als Beispiel vor, dass die Funktion `rechteckumfang` einen Fehlerstatus als Rückgabe liefern soll und darüber hinaus bei gültigen Argumentwerten auch den Rechteckumfang.

Fragen

- Das Prinzip der Werteübergabe durch Kopieren (call by value) scheint leicht nutzbar zu sein, wenn es in jeder Funktion *nur einen Rückgabewert* als Funktionswert gibt.
- Es sind aber Funktionen denkbar, die *mehr als einen Wert zurückgeben.*
- Stellen wir uns als Beispiel vor, dass die Funktion `rechteckumfang` einen Fehlerstatus als Rückgabe liefern soll und darüber hinaus bei gültigen Argumentwerten auch den Rechteckumfang.
- Um ein solches Szenario vorteilhaft in C zu implementieren, übergibt man *Adressen* anstelle der Variablenwerte. Dadurch wird in den Original-Speicherbereichen der aufrufenden Funktion operiert.

Referenzieren und Dereferenzieren

Wir benötigen dazu zwei wichtige Operatoren:

Referenzieren

Adressoperator `&`:
Sei `<Typ> a;` deklariert, dann liefert `&a` die Anfangsadresse von `a` im Speicher.
(Referenz: Verweis auf einen Speicherbereich)

Referenzieren und Dereferenzieren

Wir benötigen dazu zwei wichtige Operatoren:

Referenzieren

Adressoperator `&`:
Sei `<Typ> a;` deklariert, dann liefert `&a` die Anfangsadresse von `a` im Speicher.
(Referenz: Verweis auf einen Speicherbereich)

Dereferenzieren

Inhaltsoperator `*`:
Sei `<Typ> *p;` deklariert, wobei `p` eine Adresse ist. Dann liefert `*p` den Variablenwert der Bitkette ab Adresse `p`. Über den Typ `<Typ>` ist festgelegt, wieviele aufeinanderfolgende Bytes im Speicher ausgelesen werden und wie die Dekodierung der Bitkette in den Variablenwert erfolgt.

Funktion zur Berechnung des Rechteckumfangs

rechteckumfang2.c

```c
#include <stdio.h>

int rechteckumfang(float a, float b, float *u)
{
  if ((a < 0) || (b < 0))
  {
    return 1;   //frei gewaehlter Fehlerstatus bei ungueltigen Seitenlaengen
  }
  *u = 2*(a+b); //u ist eine Adresse, *u ihr Inhalt als Variablenwert
  return 0;     //frei gewaehlter Fehlerstatus bei erfolgreicher Abarbeitung
}


int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;
  int s = 0;

  s = rechteckumfang(x, y, &z); //&z ist die Adresse, ab der z im Speicher abgelegt ist
  if (s == 0)
  {
    printf("Der Umfang betraegt: %f\n", z); //Ausgabe nur bei erfolgreicher Berechnung
  }
  return 0;
}
```

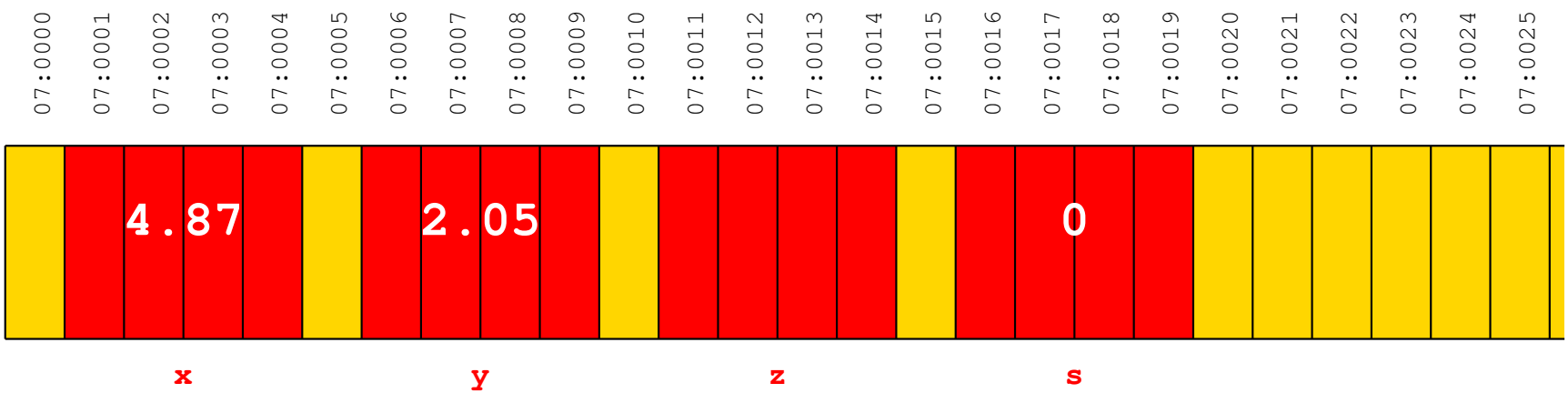
Übergabe einer Adresse als Parameterwert

```c
#include <stdio.h>

int rechteckumfang(float a, float b, float *u)
{
  if ((a < 0) || (b < 0))
  {
    return 1;
  }
  *u = 2*(a+b);
  return 0;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;
  int s = 0;
  s = rechteckumfang(x, y, &z);
  if (s == 0)
  {
    printf("Umfang: %f\n", z);
  }
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Durch Übergabe einer Adresse kann an den *Originalspeicherplätzen* von Variablen der aufrufenden Funktion operiert werden.

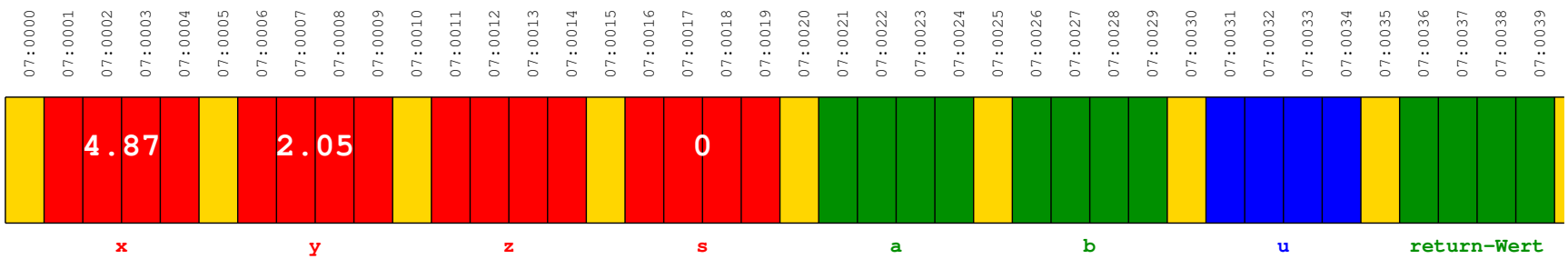
Übergabe einer Adresse als Parameterwert

```c
#include <stdio.h>

int rechteckumfang(float a, float b, float *u)
{
 if ((a < 0) || (b < 0))
 {
  return 1;
 }
 *u = 2*(a+b);
 return 0;
}
```

```c
int main(void)
{
 float x = 4.87;
 float y = 2.05;
 float z;
 int s = 0;
 s = rechteckumfang(x, y, &z);
 if (s == 0)
 {
  printf("Umfang: %f\n", z);
 }
 return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Bei Funktionsaufruf für jedes Argument eine entsprechende Variable angelegt und dafür *neuer Speicherplatz* reserviert, hier also für die **`float`**-Variablen **`a`** und **`b`** sowie die *Adresse* **`u`**.

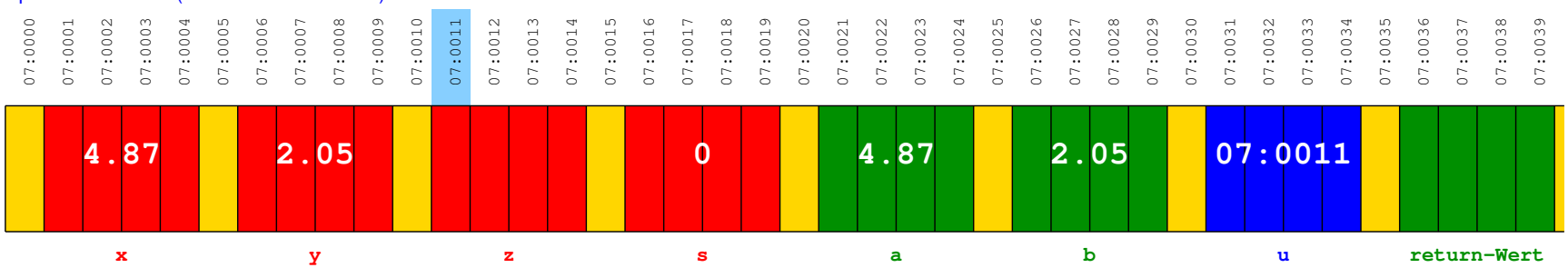
Übergabe einer Adresse als Parameterwert

```c
#include <stdio.h>

int rechteckumfang(float a, float b, float *u)
{
 if ((a < 0) || (b < 0))
 {
  return 1;
 }
 *u = 2*(a+b);
 return 0;
}
```

```c
int main(void)
{
 float x = 4.87;
 float y = 2.05;
 float z;
 int s = 0;
 s = rechteckumfang(x, y, &z);
 if (s == 0)
 {
  printf("Umfang: %f\n", z);
 }
 return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



float-Werte von **x** und **y** werden in die Speicherbereiche von **a** und **b** kopiert. Darüber hinaus wird die *Adresse von* **z** in den Speicherbereich von **u** geschrieben.

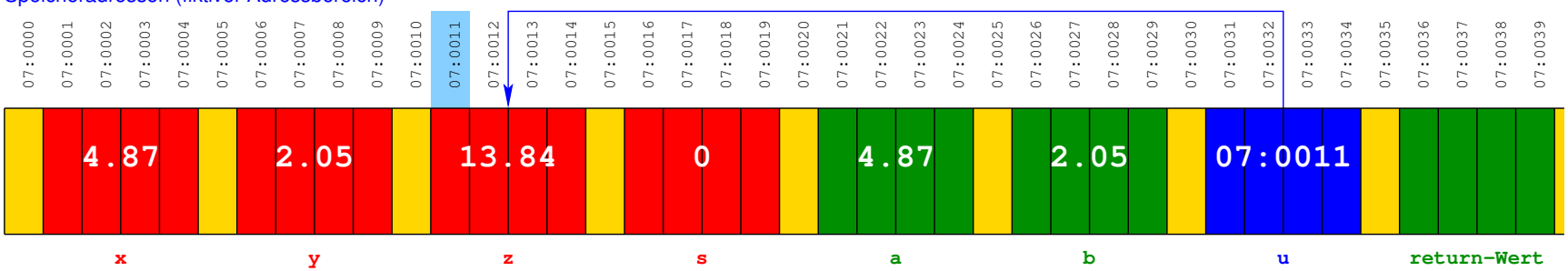
Übergabe einer Adresse als Parameterwert

```c
#include <stdio.h>

int rechteckumfang(float a, float b, float *u)
{
  if ((a < 0) || (b < 0))
  {
    return 1;
  }
  *u = 2*(a+b);
  return 0;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;
  int s = 0;
  s = rechteckumfang(x, y, &z);
  if (s == 0)
  {
    printf("Umfang: %f\n", z);
  }
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Als Inhalt `*u` der in `u` hinterlegten Adresse wird das Berechnungsergebnis `13.84` (ergibt sich aus $2 \cdot (4.87 + 2.05)$) geschrieben. Die in `u` hinterlegte Adresse verweist wie gewünscht auf den Speicherbereich der `float`-Variablen `z`.

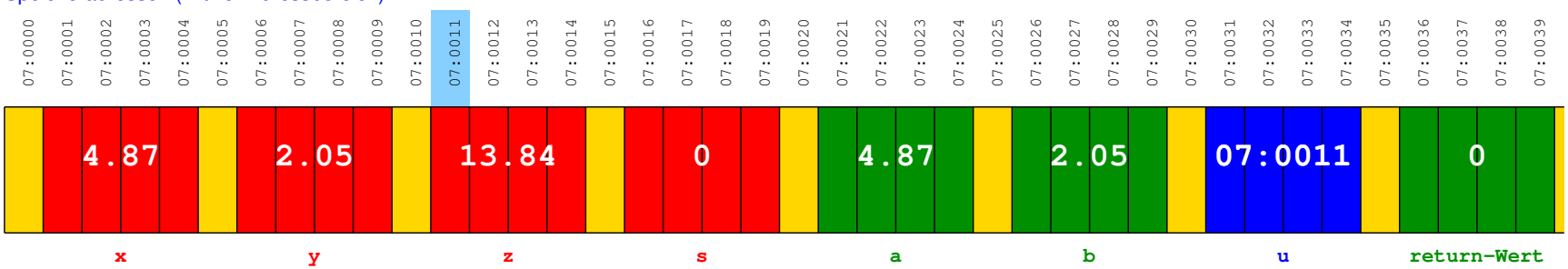
Übergabe einer Adresse als Parameterwert

```c
#include <stdio.h>

int rechteckumfang(float a, float b, float *u)
{
 if ((a < 0) || (b < 0))
 {
  return 1;
 }
 *u = 2*(a+b);
 return 0;
}
```

```c
int main(void)
{
 float x = 4.87;
 float y = 2.05;
 float z;
 int s = 0;
 s = rechteckumfang(x, y, &z);
 if (s == 0)
 {
  printf("Umfang: %f\n", z);
 }
 return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Der Rückgabewert **0** wird an den dafür vorgesehenen Speicherplatz geschrieben und in den Speicherbereich von **s** kopiert, dessen Wert sich dadurch nicht ändert.

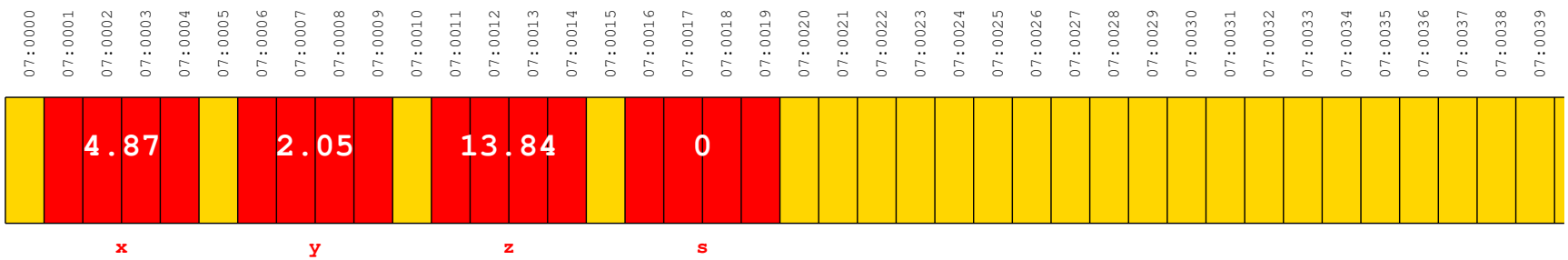
Übergabe einer Adresse als Parameterwert

```c
#include <stdio.h>

int rechteckumfang(float a, float b, float *u)
{
  if ((a < 0) || (b < 0))
  {
    return 1;
  }
  *u = 2*(a+b);
  return 0;
}
```

```c
int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;
  int s = 0;
  s = rechteckumfang(x, y, &z);
  if (s == 0)
  {
    printf("Umfang: %f\n", z);
  }
  return 0;
}
```

Speicheradressen (fiktiver Adressbereich)



Nachdem die Funktion **rechteckumfang** abgearbeitet ist, werden die von ihr beanspruchten Speicherbereiche wieder freigegeben. Der berechnete Rechteckumfang steht in der **float**-Variablen **z** zur Verfügung, während der Funktionswert in der **int**-Variablen **s** den Fehlerstatus **0** (kein Fehler) liefert.

Parameterübernahme von der Kommandozeile

mainargsrechteck.c – Zinseszinsberechnung

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>  //macht die Funktionen atof und atoi verfuegbar

int main(int argc, char* argv[])
{
  float startkapital, zinssatz;
  int laufzeit;

  if (argc != 4)               //Anzahl eingelesene Argumente einschl. exe-Dateiname
  {
    printf("Ungueltige Eingabe!\n");
    return -1;
  }
                               //argv[0] ist Name der exe-Datei
  startkapital = atof(argv[1]); //atof wandelt Zeichenkette in float-Wert um
  zinssatz = atof(argv[2]);
  laufzeit = atoi(argv[3]);     //atoi wandelt Zeichenkette in int-Wert um

  printf("Endkapital: %.2f\n", pow(1.0+zinssatz/100.0,laufzeit)*startkapital);
  return 0;
}

//Aufruf: a.exe <Startkapital> <jaehrl_Zinssatz_Prozent> <Laufzeit_in_Jahren>
//Aufrufparameter stets durch genau ein Leerzeichen voneinander trennen
//also z.B.: a.exe 5000.00 3.5 20
```

Parameterübernahme von der Kommandozeile

```
int main(int argc, char* argv[])
```

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Beispielaufruf: `a.exe 5000.00 3.5 20`

Parameterübernahme von der Kommandozeile

```
int main(int argc, char* argv[])
```

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Beispielaufruf: `a.exe 5000.00 3.5 20`
- `argc` liefert *Anzahl* eingelesener Parameter
- Name der gestarteten Programmdatei zählt dabei mit (im obigen Beispiel also 4 Parameter)

Parameterübernahme von der Kommandozeile

```
int main(int argc, char* argv[])
```

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Beispielaufruf: `a.exe 5000.00 3.5 20`
- `argc` liefert *Anzahl* eingelesener Parameter
- Name der gestarteten Programmdatei zählt dabei mit (im obigen Beispiel also 4 Parameter)
- `argv[]` ist ein Feld von Zeichenketten. Jeder Parameter als Zeichenkette bereitgestellt
- Die einzelnen Zeichenketten haben die Namen `argv[0]`, `argv[1]`, `argv[2]`, `argv[3]`

Parameterübernahme von der Kommandozeile

```
int main(int argc, char* argv[])
```

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Beispielaufruf: `a.exe 5000.00 3.5 20`
- `argc` liefert *Anzahl* eingelesener Parameter
- Name der gestarteten Programmdatei zählt dabei mit (im obigen Beispiel also 4 Parameter)
- `argv[]` ist ein Feld von Zeichenketten. Jeder Parameter als Zeichenkette bereitgestellt
- Die einzelnen Zeichenketten haben die Namen `argv[0]`, `argv[1]`, `argv[2]`, `argv[3]`

Bibliotheksfunktionen `atof` und `atoi` (über `stdlib.h` verfügbar)

- `atof` wandelt Zeichenkette in `double`- bzw. `float`-Wert um
- `atoi` wandelt Zeichenkette in `int`-Wert um, sofern möglich

Lange Quelltexte ermüden . . .

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <arpa/inet.h>

void serveur1(portServ ports)
{
    int sockServ1, sockServ2, sockClient;
    struct sockaddr_in monAddr, addrClient, addrServ2;
    socklen_t lenAddrClient;

    if ((sockServ1 = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("Erreur socket");
    exit(1);
    }
    if ((sockServ2 = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("Erreur socket");
    exit(1);
    }

    bzero(&monAddr, sizeof(monAddr));
    monAddr.sin_family = AF_INET;
    monAddr.sin_port = htons(ports.port1);
    monAddr.sin_addr.s_addr = INADDR_ANY;
    bzero(&addrServ2, sizeof(addrServ2));
```

Abschreckendes Beispiel aus der Programmierer-Mottenkiste
(enthält einige noch nicht behandelte Sprachkonstrukte in C)

Funktionen in Quelltexten übersichtlich anordnen

- C-Programmquelltexte als Sammlung von Funktionen anzulegen, ist sinnvoll
- Wir haben unsere selbstdefinierten Funktionen stets *vor* die `main`-Funktion geschrieben
- Dies erschwert aber die Lesbarkeit *langer* Quelltexte mit *vielen* Funktionen, denn man will dann gern zuerst die `main`-Funktion sehen

Funktionen in Quelltexten übersichtlich anordnen

- C-Programmquelltexte als Sammlung von Funktionen anzulegen, ist sinnvoll
- Wir haben unsere selbstdefinierten Funktionen stets *vor* die `main`-Funktion geschrieben
- Dies erschwert aber die Lesbarkeit *langer* Quelltexte mit *vielen* Funktionen, denn man will dann gern zuerst die `main`-Funktion sehen
- Übersichtlich wäre, wenn im Quelltext als erstes die `main`-Funktion steht
- Schreiben wir unsere selbstdefinierten Funktionen *hinter* die `main`-Funktion, erleben wir eine böse Überraschung beim Compilieren (sofern der ANSI-C-Standard verwendet wird): Fehlermeldungen, dass unsere Funktionen *unbekannt* seien

Funktions-Prototypen angeben

„Um dem C-Compiler im ersten Auswertungslauf durch den Quelltext die erwarteten Funktionen bekanntzumachen, schreiben wir an den Quelltextanfang unmittelbar hinter den Präprozessorteil untereinander alle *Funktionsköpfe* jeweils durch Semikolon getrennt.“

Funktions-Prototypen angeben

rechteckumfang3.c – Prototyp der Funktion `rechteckumfang` vor `main`-Funktion

```c
#include <stdio.h>

float rechteckumfang(float a, float b); //Funktions-Prototyp


int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}


float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

Argumentnamen darf man in Prototypen weglassen

rechteckumfang4.c

```c
#include <stdio.h>

float rechteckumfang(float, float); //Funktions-Prototyp


int main(void)
{
  float x = 4.87;
  float y = 2.05;
  float z;

  z = rechteckumfang(x, y);
  printf("Der Umfang betraegt: %f\n", z);
  return 0;
}


float rechteckumfang(float a, float b)
{
  float u = 2*(a+b);
  return u;
}
```

Modularisierung – Softwaremodule bauen

Idee der Modularisierung

- Zum Zubereiten einer guten Mahlzeit nutzt man viele, teilweise vorgefertigte Zutaten
- Diese Zutaten sind *Bausteine*, aus denen sich die *Mahlzeit zusammensetzt*
- Die Zutaten ergänzen sich sinnvoll in der Mahlzeit und sind weitgehend unabhängig voneinander
- *Funktionen* sind die Zutaten für eine *Funktionsbibliothek*, auch *Modul* genannt

Begriff „Modul“

Ein **Modul** ist eine abgeschlossene *funktionelle Einheit* einer Software. Es umfasst entwicklungstechnisch eigenständige Programmteile (z.B. Funktionen), die über eine modulspezifische Schnittstelle aufgerufen und mit Daten versorgt werden (z.B. Parameterübergabe durch Funktionsargumente). Ein Modul kann selbst weitere Module einbinden und Funktionen daraus aufrufen, so dass eine Modulhierarchie sichtbar wird.

Begriff „Modul“

Ein **Modul** ist eine abgeschlossene *funktionelle Einheit* einer Software. Es umfasst entwicklungstechnisch eigenständige Programmteile (z.B. Funktionen), die über eine modulspezifische Schnittstelle aufgerufen und mit Daten versorgt werden (z.B. Parameterübergabe durch Funktionsargumente). Ein Modul kann selbst weitere Module einbinden und Funktionen daraus aufrufen, so dass eine Modulhierarchie sichtbar wird.

$\Longrightarrow$ Jede Standard-Funktionsbibliothek ist für sich genommen ein Modul (z.B. `math.h` oder `stdio.h`).

Begriff „Modul“

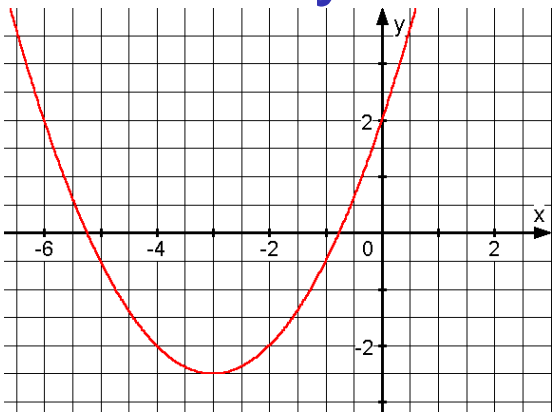
Ein **Modul** ist eine abgeschlossene *funktionelle Einheit* einer Software. Es umfasst entwicklungstechnisch eigenständige Programmteile (z.B. Funktionen), die über eine modulspezifische Schnittstelle aufgerufen und mit Daten versorgt werden (z.B. Parameterübergabe durch Funktionsargumente). Ein Modul kann selbst weitere Module einbinden und Funktionen daraus aufrufen, so dass eine Modulhierarchie sichtbar wird.

$\Longrightarrow$ Jede Standard-Funktionsbibliothek ist für sich genommen ein Modul (z.B. `math.h` oder `stdio.h`).

Wie schreibt man eigene Module?

Modul zum Handling quadratischer Polynome

Als Beispiel bauen wir uns ein Modul mit C-Funktionen auf quadratischen Polynomen



$f(x) = a_2 \cdot x^2 + a_1 \cdot x + a_0$

mit $a_2, a_1, a_0 \in \mathbb{R}$ und $a_2 \neq 0$

```
float f(float a2, float a1, float a0, float x)
```
. . . liefert $f(x)$

```
int hatReelleNst(float a2, float a1, float a0)
```
. . . liefert 1, falls reelle Nullstelle vorhanden, sonst 0

```
float reelleNst1(float a2, float a1, float a0)
```
. . . liefert erste Nullstelle

```
float reelleNst2(float a2, float a1, float a0)
```
. . . liefert zweite Nullstelle

```
float scheitelx(float a2, float a1, float a0)
```
. . . liefert x-Koordinate des Scheitelpunkts

```
float scheitely(float a2, float a1, float a0)
```
. . . liefert y-Koordinate des Scheitelpunkts

Zunächst alles in einer C-Quelltextdatei

quadratisches-polynom.c – Teil 1

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

/* Prototypen */

float f(float a2, float a1, float a0, float x);
int hatReelleNst(float a2, float a1, float a0);
float reelleNst1(float a2, float a1, float a0);
float reelleNst2(float a2, float a1, float a0);
float scheitelx(float a2, float a1, float a0);
float scheitely(float a2, float a1, float a0);

/* main mit Testrahmen */

int main(void)
{
  float p, q;

  printf("Quadratisches Polynom f(x) = x*x + p*x + q\n\n");
  printf("Bitte Koeffizienten p eingeben: ");
  scanf("%f", &p);
  printf("Bitte Koeffizienten q eingeben: ");
  scanf("%f", &q);

  if (hatReelleNst(1, p, q))
  {
    printf("Nullstellen: %f und %f\n", reelleNst1(1, p, q), reelleNst2(1, p, q));
  }
  printf("Scheitelpunkt: (%f, %f)\n", scheitelx(1, p, q), scheitely(1, p, q));
  return 0;
}
```

Zunächst alles in einer C-Quelltextdatei

quadratisches-polynom.c – Teil 2

```c
/* selbstdefinierte Funktionen */

float f(float a2, float a1, float a0, float x)
{
  return a2*x*x + a1*x + a0;
}

int hatReelleNst(float a2, float a1, float a0)
{
  return (a1*a1/4 >= a0);
}

float reelleNst1(float a2, float a1, float a0)
{
  if (hatReelleNst(a2, a1, a0))
  {
    return -a1/(2*a2) - sqrt(a1*a1/(4*a2) - a0/a2);
  }
  return -1.7e+38; //Rueckgabewert im Fehlerfall
}
```

```c
float reelleNst2(float a2, float a1, float a0)
{
  if (hatReelleNst(a2, a1, a0))
  {
    return -a1/(2*a2) + sqrt(a1*a1/(4*a2) - a0/a2);
  }
  return -1.7e+38; //Rueckgabewert im Fehlerfall
}

float scheitelx(float a2, float a1, float a0)
{
  return -a1/(2*a2);
}

float scheitely(float a2, float a1, float a0)
{
  return f(a2, a1, a0, scheitelx(a2, a1, a0));
}
```

Prototypen als Headerdatei (*.h) auslagern

qpol.h

```
#ifndef QPOL
#define QPOL   //verhindert Mehrfach-Einbindungen

float f(float, float, float, float);
int hatReelleNst(float, float, float);
float reelleNst1(float, float, float);
float reelleNst2(float, float, float);
float scheitelx(float, float, float);
float scheitely(float, float, float);

#endif
```

- Prototypen der Funktionen im Modul als *Headerdatei* (Endung .h) speichern, Argumentnamen darf man weglassen
- Präprozessorkonstrukt **`#ifndef ... #endif`** verhindert Mehrfach-Einschluss
- Mehrfach-Einschluss: mehrmaliges Einbinden der gleichen Headerdatei $\rightarrow$ Compilierfehler

Funktionsdefinitionen des Moduls als .c-Datei

qpol.c

```c
#include <math.h>

float f(float a2, float a1, float a0, float x)
{
  return a2*x*x + a1*x + a0;
}

int hatReelleNst(float a2, float a1, float a0)
{
  return (a1*a1/4 >= a0);
}

float reelleNst1(float a2, float a1, float a0)
{
  if (hatReelleNst(a2, a1, a0))
  {
    return -a1/(2*a2) - sqrt(a1*a1/(4*a2) - a0/a2);
  }
  return -1.7e+38; //Rueckgabewert im Fehlerfall
}
```

```c
float reelleNst2(float a2, float a1, float a0)
{
  if (hatReelleNst(a2, a1, a0))
  {
    return -a1/(2*a2) + sqrt(a1*a1/(4*a2) - a0/a2);
  }
  return -1.7e+38; //Rueckgabewert im Fehlerfall
}

float scheitelx(float a2, float a1, float a0)
{
  return -a1/(2*a2);
}

float scheitely(float a2, float a1, float a0)
{
  return f(a2, a1, a0, scheitelx(a2, a1, a0));
}
```

- Funktionsdefinitionen des Moduls: .c-Datei gleichnamig zur .h
- Guter Programmierstil: keine Bildschirmausgaben und keine Tastatureingaben in den Funktionen des Moduls vorsehen

Gesonderte .c-Datei mit `main`-Funktion

quadratisches-polynom2.c

```
Quadratisches Polynom f(x) = x*x + p*x + q

Bitte Koeffizienten p eingeben: -6
Bitte Koeffizienten q eingeben: 8
Nullstellen: 2.000000 und 4.000000
Scheitelpunkt: (3.000000, -1.000000)
```

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

#include "qpol.h" // unsere Header-Datei des Moduls

int main(void)
{
  float p, q;

  printf("Quadratisches Polynom f(x) = x*x + p*x + q\n\n");
  printf("Bitte Koeffizienten p eingeben: ");
  scanf("%f", &p);
  printf("Bitte Koeffizienten q eingeben: ");
  scanf("%f", &q);

  if (hatReelleNst(1, p, q))
  {
    printf("Nullstellen: %f und %f\n", reelleNst1(1, p, q), reelleNst2(1, p, q));
  }
  printf("Scheitelpunkt: (%f, %f)\n", scheitelx(1, p, q), scheitely(1, p, q));
  return 0;
}
```

Compilieren: **`gcc qpol.c quadratisches-polynom2.c -lm`**

C-Standardbibliotheken als vorcompilierte Module

`assert.h`	Funktionsprototypen zur Programmdiagnose
`ctype.h`	Funktionen und Makros zur zeichenweisen Bearbeitung
`errno.h`	beinhaltet Fehlernummern
`float.h`	enthält Fließkomma-Grenzwerte
`iso646.h`	enthält eine Reihe von Makros für Operatoren
`limits.h`	enthält Ganzzahl-Grenzwerte
`locale.h`	Anpassung an spezielle nationale Gegebenheiten
`math.h`	mathematische Deklarationen und Routinen
`setjmp.h`	globale Sprünge
`signal.h`	Funktionen zur Signalverarbeitung
`stdarg.h`	Arbeiten mit variablen Argumentlisten
`stddef.h`	Deklaration allgemeiner Werte
`stdio.h`	Standard-Ein-und-Ausgabe
`stdlib.h`	Hilfsfunktionen und allgemeine Konstanten
`string.h`	Zeichenkettenverarbeitung
`time.h`	Datum und Uhrzeit