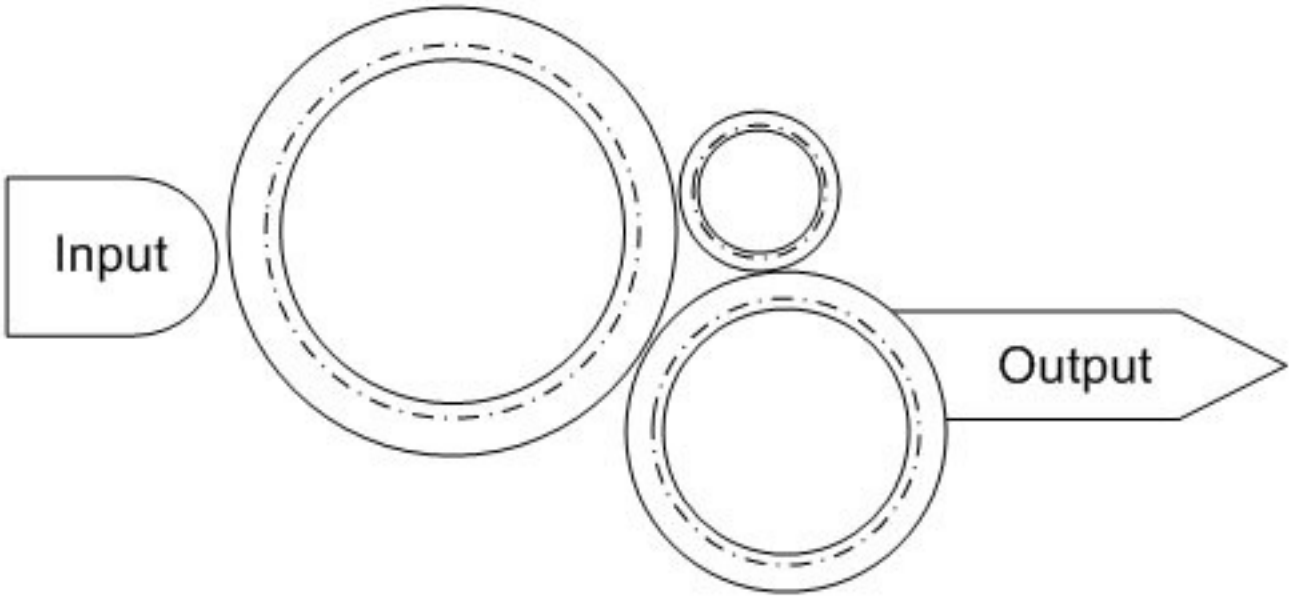
C++ - Einführung in die Programmiersprache

Subroutine

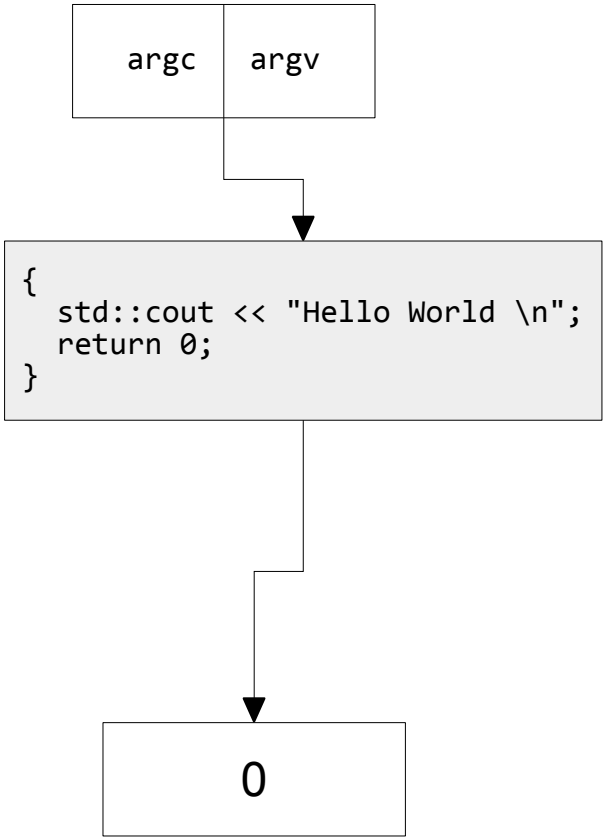


Subroutinen

- Strukturierung von großen Codeabschnitten.
- Beschreibung einer Handlung in einem Codeblock.
- Wiederkehrende Tätigkeiten werden zusammengefasst.
- Wiederverwendung von Codeblöcken.

Arbeitsweise von Subroutinen

- Subroutinen entsprechen einer Fertigungsstraße in einer Fabrikhalle.
- Der Betrachter weiß, wie die Fertigung gestartet werden muss. Eventuell wird Material zum Start benötigt.
- Bei einigen Subroutinen wird dem Betrachter das fertige Werkstück am Ende ausgeliefert. Bei anderen Subroutinen verbleibt es in der Fabrik.
- Wie das Werkstück erstellt wird, sieht der Betrachter nicht.



Bottom-up-Konstruktion

- Beginnend mit dem Möglichkeiten, die die Programmiersprache C++ bietet, werden kleine Teilaufgaben gelöst.
- Diese Teilaufgaben werden in x Schritte so weit vergrößert, so dass am Schluss die Aufgabenstellung im Programm abgebildet wird.
- Hinweis: Der geschriebene Programmiercode muss jederzeit in einem anderen Projekt wieder verwendet werden.

Top-Down-Konstruktion

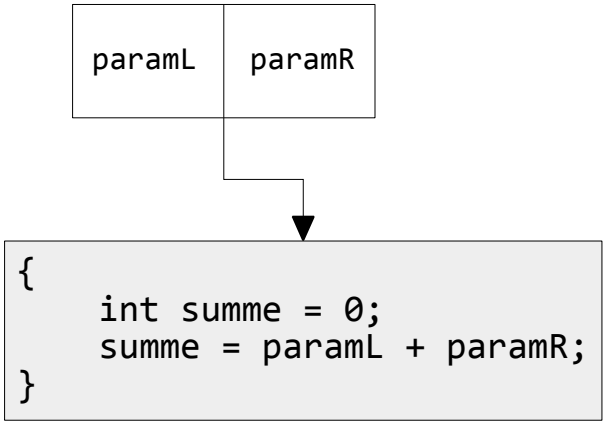
- Die zu lösende Aufgabenstellung wird in x Schritten in immer kleinere Teilaufgaben unterteilt.
- Zum Schluss sind die Teilaufgaben so klein, dass diese mit Hilfe von Subroutinen in der Programmiersprache C++ abgebildet werden können.

Prozeduren

```
void addition(int paramL, int paramR){
    int summe = 0;
    summe = paramL + paramR;
}

addition(argL, argR);
```

Arbeitsweise

- Der Benutzer kennt die Signatur (Funktionskopf). In diesem Beispiel wird ein Taschenrechner simuliert.
- Der Betrachter weiß, wie die der Taschenrechner gestartet werden muss.
- In diesem Beispiel benötigt der Taschenrechner für den Start der Berechnung zwei Werte.
- Das Ergebnis der Berechnung wird dem Benutzer nicht mitgeteilt.

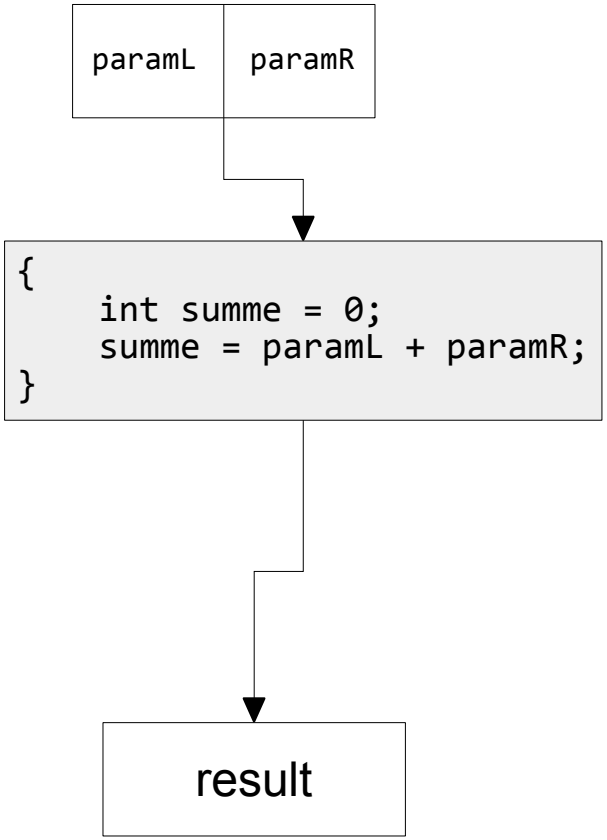


Funktionen

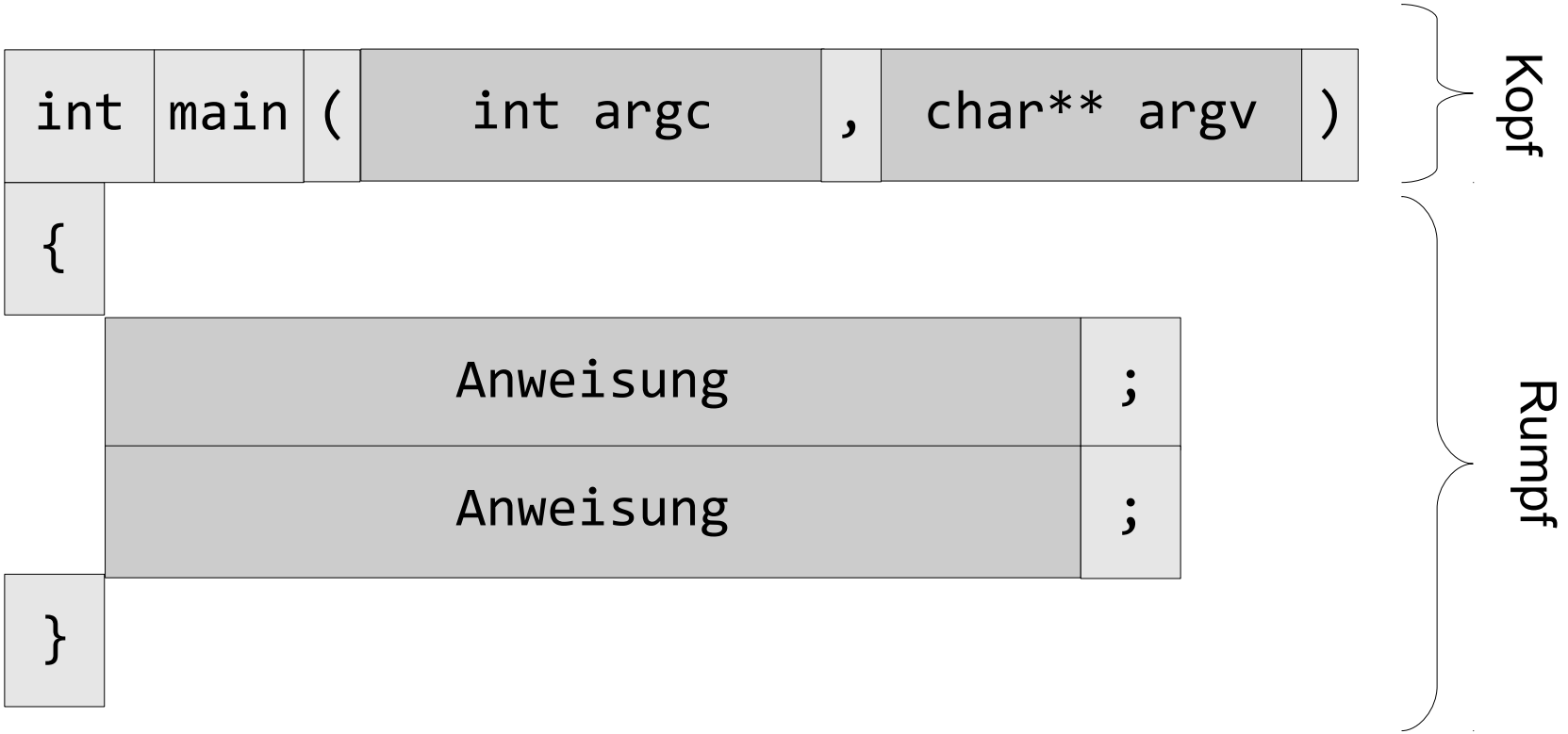
```
int addition(int paramL, int paramR){
    return(paramL + paramR);
}

result = addition(argL, argR);
```

Arbeitsweise

- Der Benutzer kennt die Signatur (Funktionskopf). In diesem Beispiel wird ein Taschenrechner simuliert.
- Der Betrachter weiß, wie die der Taschenrechner gestartet werden muss.
- In diesem Beispiel benötigt der Taschenrechner für den Start der Berechnung zwei Werte.
- Das Ergebnis der Berechnung wird dem Benutzer auf dem Display mitgeteilt.



Aufbau einer Subroutine



Rumpf einer Subroutine

- Beginn und Ende mit dem geschweiften Klammern.
- In dem Codeblock wird eine Handlung beschrieben.
- In dem Rumpf der Subroutine kann keine weitere Subroutine definiert werden. Aber Subroutinen können aufgerufen werden.

Rumpf der Funktion main

```
int main(int argc, char** argv) {
    std::cout << "Hello World \n";
    return 0;
}
```

- Startzentrale eines C++-Programms.
- Die Funktion `main()` startet den Fertigungsprozess und verzweigt entsprechend der gestellten Aufgabe.
- Verzweigung in verschiedene Funktionen oder Prozeduren.

Kopf einer Subroutine

- Signatur der Subroutine.
- Schnittstelle zu einer Subroutine nach außen.

Aufbau des Kopfes

- Datentyp der Subroutine. Welchen Typ von Ergebnis liefert die Subroutine? Wird ein Ergebnis zurückgeliefert?
- Der Name identifiziert eine Subroutine. Aufruf mit Hilfe des Namens.
- Parameterliste, begrenzt durch die runden Klammern. Liste von x beliebig vielen Startwerten.

Name der Startfunktion

- Beim Aufruf eines C++ - Programms wird immer die Funktion `main` automatisiert gestartet.
- Der Bezeichner muss exakt so geschrieben. Die Programmiersprache entscheidet zwischen Groß- und Kleinschreibung.

Parameterliste

- Die Parameterliste folgt direkt dem Name der Subroutine.
- Die Liste wird durch die runden Klammern begrenzt.
- Die Liste kann beliebig viele Parameter unterschiedlichsten Typs enthalten.
- Die Parameterliste kann leer sein. Im Rumpf der Subroutine sind alle benötigten „Materialien“ deklariert.

… der Funktion main

- Die Liste kann leer sein, muss aber nicht.
- Bei dem Aufruf der ausführbaren Datei können der Funktion `main` Werte übergeben werden.

Parameter

- Übergabe von Startwerten an eine Subroutine.
- Parameter werden wie Variablen deklariert. Der Datentyp legt den Speicherbedarf des Parameters fest. Der Name ist ein eindeutiger Platzhalter für diese Speicherstelle. An der Speicherstelle wird ein Wert gespeichert.
- Die Parameter in der Liste werden durch ein Komma getrennt.

… der Funktion main

```
int | main | ( | int argc | , | char** argv | )
```

- `argc` (argument count) enthält die Anzahl der Argumente. Der Wert des Parameters kann 0 sein.
- `argv` (argument vector) ist eine Liste von Parametern vom Datentyp `string`. Die zwei Sterne im Anschluss an den Datentyp `char` verweist auf einen Zeiger, der wiederum auf einen Zeiger vom Datentyp zeigt.

Beispiel: Keine Übergabe von Argumenten

programm

```
{

    std::cout << "Anzahl der Argumente: " << argc << std::endl;

    for(int i = 0; i < argc; i++){
        std::cout << i << ". Argumente: " << argv[i] << std::endl;
    }
    return 0;
}
```

Anzahl der Argumente: 1
0. Argument: programm

Beispiel: Übergabe von x Argumenten

programm a b

```
{

    std::cout << "Anzahl der Argumente: " << argc << std::endl;

    for(int i = 0; i < argc; i++){
        std::cout << i << ". Argumente: " << argv[i] << std::endl;
    }
    return 0;
}
```

Anzahl der Argumente: 3
0. Argument: programm
1. Argument: a
2. Argument: b

Datentyp der Subroutine

- Der Kopf beginnt mit einem Datentyp.
- Die Funktion ist vom Datentyp ...
- Der Datentyp legt den Typ des Rückgabewertes der Subroutine fest.

… der Funktion main

- Die Funktion hat den Datentyp `int`.
- Mit Hilfe der Anweisung `return` wird eine Ganzzahl an den Aufrufer übergeben.
- Der Aufrufer erhält damit Informationen, ob das Programm fehlerfrei beendet wurde oder nicht. Der Rückgabewert 0 signalisiert einen fehlerfreien Durchlauf.

Schreiben von eigenen Subroutinen

```
#include <iostream>

void ausgeben(char berechnung, int opL, int opR, int result){
    std::cout << opL << ' ' << berechnung << ' ' << opR;
    std::cout << " = " << result << std::endl;
}

int addieren(int paramL, int paramR){
    int result = paramL + paramR;

    return(result);
}
```

… und aufrufen

```
int main(){
  int result = 0;
  int argL = 4;
  int argR = 5;

  result = addieren(argL,argR);
  ausgeben('+', argL, argR, result);
  return 0;
}
```

- Die Subroutinen werden mit Hilfe ihres Namens aufgerufen.
- Dem Namen folgt die Parameterliste entsprechend der Signatur der Subroutine.

Bezeichner

- Namen für Subroutinen, Konstanten, Variablen. Feldern und so weiter.
- Namen von Subroutinen sind in einer C++-Datei in Abhängigkeit des Namensraums eindeutig.
- Schlüsselwörter der Programmiersprache sind als benutzerdefinierte Namen nicht erlaubt.
- Unterscheidung zwischen Groß- und Kleinschreibung. Der Name „addiere“ und „Addiere“ sind für den Compiler verschiedene Bezeichner.

Erlaubte Zeichen

- Buchstaben A...Z und a...z.
- Zahlen 0...9.
- Der Unterstrich.

Konventionen

- Benutzerdefinierte Namen beginnen mit einem Buchstaben.
- Namen, die mit einem Unterstrich beginnen, werden für Bezeichnungen aus dem Standard von C++, den zugelassenen Bibliotheken und Makronamen genutzt.
- Leerzeichen werden durch Unterstriche ersetzt. Zum Beispiel celsius_To_Fahrenheit.
- Namen, die nur aus Großbuchstaben und Unterstrichen bestehen, sind Makros oder Konstanten vorbehalten.

Konventionen für Subroutinen

- Häufig werden Verben genutzt, die die abgebildete Tätigkeit beschreiben.
- Subroutinen, die mit „is“ beginnen, liefern einen boolschen Wert zurück. Zum Beispiel „isInteger“ überprüft, ob der übergebene Parameter ein Integer ist.
- Subroutinen, die mit „set“ beginnen, setzen eine Variable.
- Subroutinen, die mit „get“ beginnen, liefern einen beliebigen Wert zurück.

Prozeduren

- Subroutinen, die keinen Wert an den Aufrufer zurückgeben.
- Funktionen vom Datentyp `void`.
- Das Ergebnis des Arbeitsprozesses wird im Rumpf der Prozedur gekapselt. Prozeduren haben keine Sprunganweisung (`return`) im Rumpf.

Nutzung

- Subroutinen, deren Ergebnis für den weiteren Ablauf des Prozesses nicht relevant sind.
- Ausgabe des Ergebnisses auf die Konsole oder in eine Datei.

… aufrufen

```
ausgeben('+', argL, argR, result);
```

- Der Aufruf entspricht dem Prozedurkopf.
- Die Prozedur wird mit Hilfe des Namens aufgerufen. Beim Aufruf wird die Groß- und Kleinschreibung beachtet.
- Dem Namen folgt die Argumentliste. Die Anzahl der Argumente in der Liste entspricht der Anzahl der Parameter in der Liste im Kopf der Prozedur.
- Die Argumente sollten entsprechend des Datentypes der Parameter der Prozedur interpretierbar sein.

Kopf einer Prozedur

- Beginn mit dem Datentyp `void`.
- Der Name der Prozedur ist eindeutig in der Datei.
- Dem Namen folgt die Parameterliste, begrenzt durch die runden Klammern. Die Anzahl der Parameter ist abhängig von der zu lösenden Aufgabe.

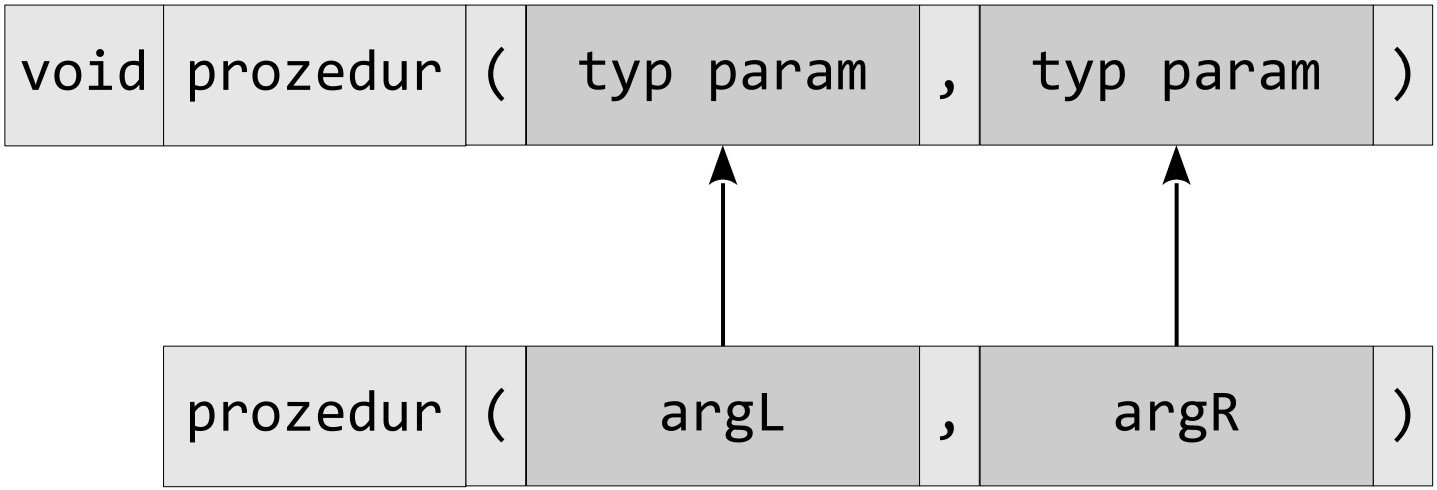
Parameterliste

- Die runden Klammern fassen x Parameter zusammen.
- Die Liste kann keine Parameter enthalten.
- Die Parameter in der Liste werden durch ein Komma getrennt.

Parameter

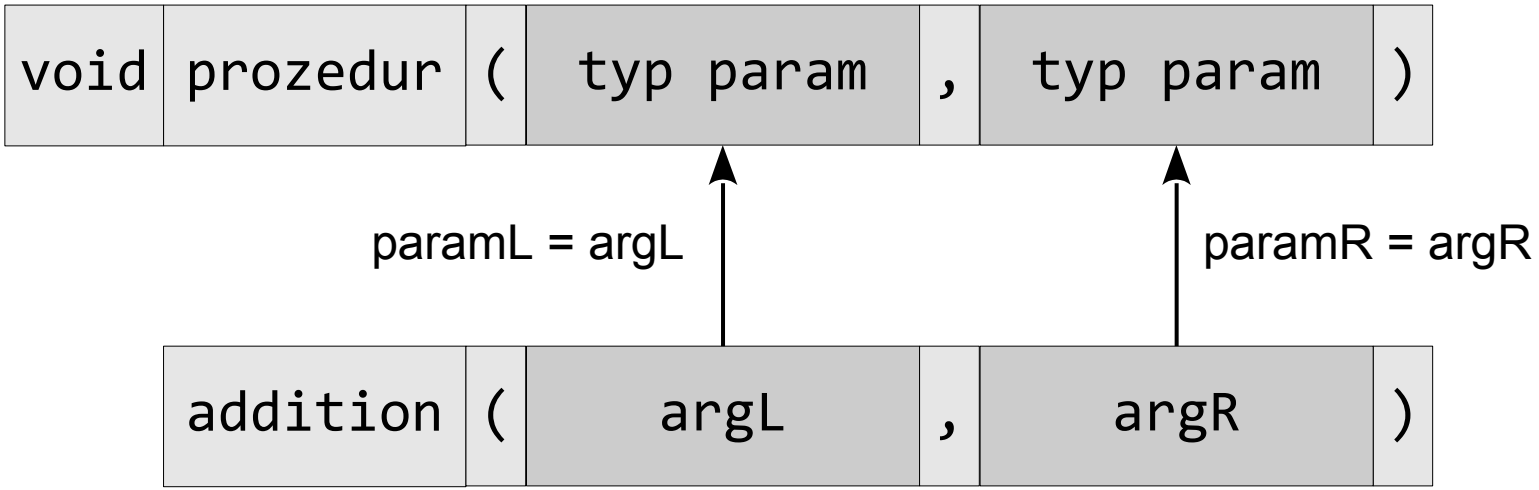
- Parameter müssen deklariert werden.
- Jeder Parameter hat einen bestimmten Datentyp. Der Datentyp legt den Wertebereich sowie den Speicherbedarf fest.
- Der Name des Parameters ist in der Prozedur eindeutig.
- Der Wert des Parameters wird durch den Aufruf der Prozedur festgelegt.

Übergabe der Parameter beim Aufruf



- Die Argumente werden den Parametern in Abhängigkeit ihrer Position zugeordnet.
- Der Wert des Arguments muss entsprechend des Datentyps des Parameters interpretierbar sein.

Call by Value



- Die Übergabe der Argumente an die Parameter entspricht einer Zuweisung.
- Der Wert des Arguments wird in den Wert des Parameters kopiert.

Funktionen

- Subroutinen, die einen Wert an den Aufrufer mit Hilfe der Anweisung `return` zurückgeben.
- Funktionen vom einem x-beliebigen Datentyp.
- Das Ergebnis des Arbeitsprozesses wird im Rumpf der Funktion nicht gekapselt. Funktionen geben eine Mitteilung an den Auftraggeber zurück.

Nutzung

- Ergebnis, die im weiteren Verlauf des Prozesses, weiter verarbeitet werden.
- Überprüfung von Werten. Hat die Variable den Status?
- Einlesen von Werten von der Konsole oder einer Datei.

Beispiel

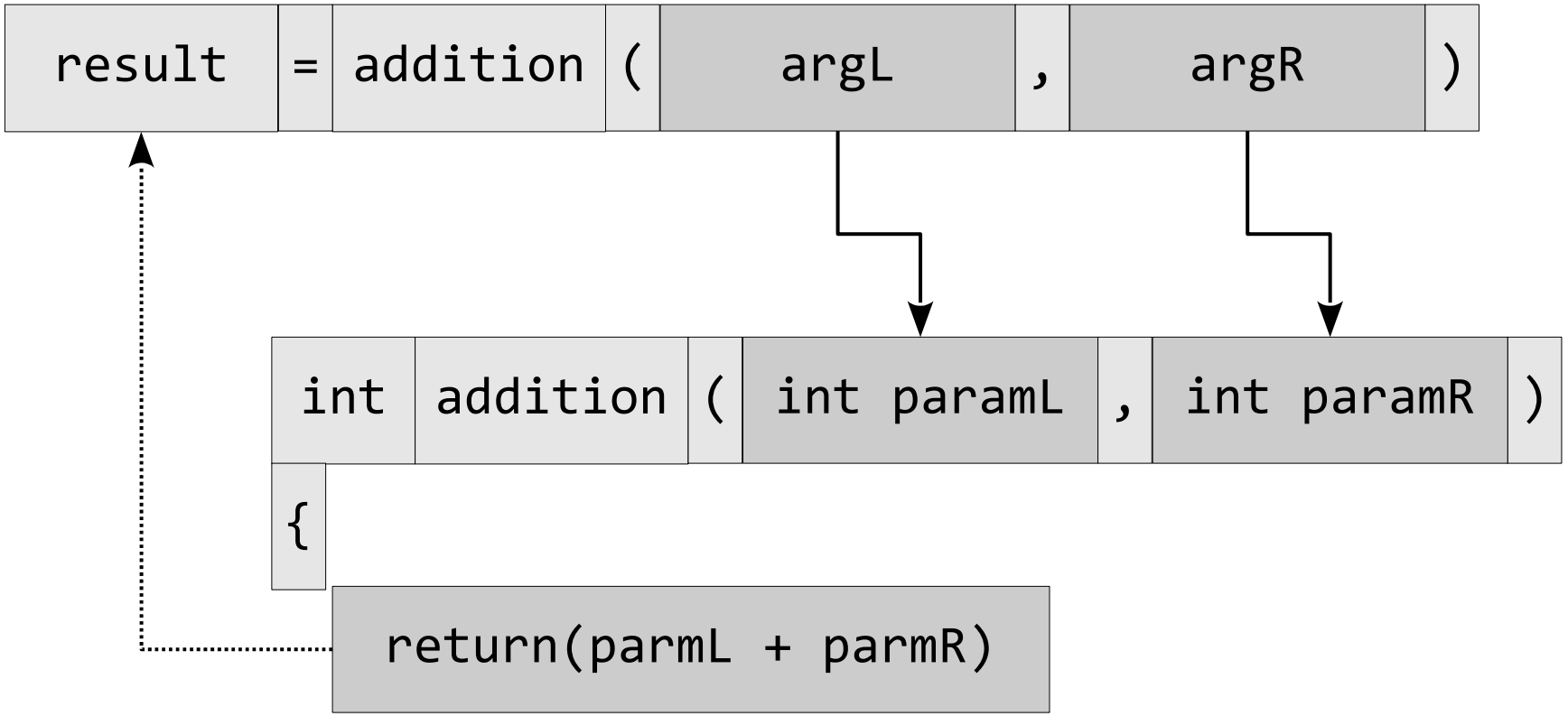
```
int addieren(int paramL, int paramR){
  int result = paramL + paramR;

  return(result);
}
```

… aufrufen

```
result = addieren(argL,argR);
```

- Der Aufruf entspricht dem Funktionskopf.
- Die Funktion wird mit Hilfe des Namens aufgerufen. Beim Aufruf wird die Groß- und Kleinschreibung beachtet.
- Dem Namen folgt die Argumentliste. Die Anzahl der Argumente in der Liste entspricht der Anzahl der Parameter in der Liste im Kopf der Funktion.
- Der Rückgabewert wird zur Weiterverarbeitung in einer Variable gespeichert.

Beispiel


result = addition (argL , argR)
int addition (int paramL , int paramR)
{
return(parmL + parmR)

Kopf einer Funktion

- Von welchem Typ ist die Funktion? Was wird zurückgegeben?
- Dem Funktionstyp folgt der Name der Funktion. Der Name ist eindeutig in der Codedatei
- Dem Namen folgt die Parameterliste.

Rückgabewert einer Funktion

```
return(parmL + parmR)
```

- Mit Hilfe von `return` wird exakt ein Wert an den Aufrufer zurückgegeben.
- Der Wert kann entsprechend des Datentyps der Funktion interpretiert werden.
- Der Wert kann direkt angegeben werden. Der Wert kann mit Hilfe einer Ausdrucksanweisung berechnet werden.

… in Abhängigkeit von Bedingungen

```
if(paramR > 0){
  return (paramL / paramR);
}
else{
  std::cout << "Division durch Null." << std::endl;
  return 0;
}
```

Lebensdauer einer Variablen

- Wie lange existiert eine Variable?
- Die Lebensdauer kann sich auf das Modul (die Datei), die Subroutine oder einen Anweisungsblock beziehen.
- Sobald eine Variable deklariert wird, wird Speicher entsprechend des Datentyps angefordert. Wenn der Codeblock verlassen wird, wird dieser automatisiert freigegeben.

Globale Variablen und Konstanten

```
const char FAHRENHEIT = 'F';
const char KELVIN = 'K';
const char CELSIUS = 'C';

double celsiusUmrechnen(double celsius, char umrechnenIn){
    double temperatur;

    switch(umrechnenIn){
        case FAHRENHEIT:
            temperatur = celsius * 1.8 + 32;
            break;

        case KELVIN:
            temperatur = celsius + 273.15;
            break;
    }
}
```

Erläuterung

- Die Variable oder Konstante ist in der gesamten Programmdatei bekannt und wird in verschiedenen Subroutinen benötigt.
- Am Anfang eines Programms kommen die Präprozessoranweisungen, der Hinweis auf den genutzten Namensraum und dann die Deklarationsanweisungen für die globalen Variablen und Konstanten.
- Wenn möglich, sollte auf globale Variablen verzichtet werden.

Lokale Variablen und Konstanten

```
const char FAHRENHEIT = 'F';
const char KELVIN = 'K';
const char CELSIUS = 'C';

double celsiusUmrechnen(double celsius, char umrechnenIn){
    double temperatur;

    switch(umrechnenIn){
        case FAHRENHEIT:
            temperatur = celsius * 1.8 + 32;
            break;

        case KELVIN:
            temperatur = celsius + 273.15;
            break;
    }
}
```

Erläuterung

- Deklarationsanweisungen in einem Codeblock.
- Parameter in einem Funktionskopf.
- Anweisungen in einem Schleifenkopf.
- Die Variablen und Konstanten können nur in dem Codeblock genutzt werden, in dem sie deklariert sind.
- Sobald der Codeblock verlassen wird, werden die Variablen aus dem Speicher entfernt.

Sichtbarkeit einer Variablen

- Wann ist ein Zugriff auf die Variable möglich?
- In welchen Bereich kann der Inhalt / den Wert der Variablen gelesen und verändert werden?
- Globale Variablen können von lokalen Variablen gleichen Namens überdeckt werden.

Nutzung der globalen Variablen

```
const char FAHRENHEIT = 'F';
const char KELVIN = 'K';
const char CELSIUS = 'C';
double temperatur = 0;


int main()
{
  temperatur = celsiusUmrechnen(4, KELVIN);
  return 0;
}
```

Überdeckung der globalen Variablen

```
const char FAHRENHEIT = 'F';
const char KELVIN = 'K';
const char CELSIUS = 'C';
double temperatur = 0;

double celsiusUmrechnen(double celsius,
                        char umrechnenIn){

    double temperatur;

    switch(umrechnenIn){
```

Nutzung der globalen Variablen

```
::temperatur = celsius * 1.8 + 32;
```

- Der Gültigkeitsoperator vor einem Bezeichner zeigt an, dass die überdeckte globale Variable genutzt werden soll.

Statische Variablen

```
int  addiere(int value){
    static int summe = 0;

    summe = summe + value;
    return(summe);
}

int main() {
    int result = 1;

    do{
        result = addiere(result);
        std::cout << "Summe: " << result << '\n';

    }while(result < 10);


    return 0;
}
```

Lebensdauer

```
static int summe = 0;
```

- Beim ersten Aufruf der Prozedur werden statische Variablen deklariert.
- Beim Verlassen der Prozedur werden die statischen Variablen nicht aus dem Speicher entfernt. Sie bleiben so lange erhalten, wie das Programm existiert.

Deklaration von statischen Variablen

`static`	`int`	`summe`		`;`
`static`	`Datentyp`	`Name`		`;`

- Die Deklaration beginnt mit dem Schlüsselwort `static`.
- Dem Schlüsselwort folgt der Datentyp der Variablen. In diesem Beispiel wird eine Ganzzahl (`int`) als Datentyp genutzt.
- Dem Datentyp folgt der Name der Variablen.

… und Initialisierung

	`summe`	`=`	`0`	`;`
	`Name`	`=`	`Wert`	`;`

- Der statischen Variablen wird mit Hilfe des Gleichheitszeichen ein Wert zugewiesen.
- Links vom Gleichheitszeichen steht der Name der zu initialisierenden Variablen.
- Der Initialisierungswert der statischen Variablen wird rechts vom Gleichheitszeichen angegeben.

Initialisierungswert

	`summe`	`=`	`0`	`;`
	`Name`	`=`	`Wert`	`;`

- Der Initialisierungswert kann entsprechend des Datentyps der zu initialisierenden Variablen interpretiert werden.
- Der statischen Variablen wird ein definierter Anfangswert zugewiesen.