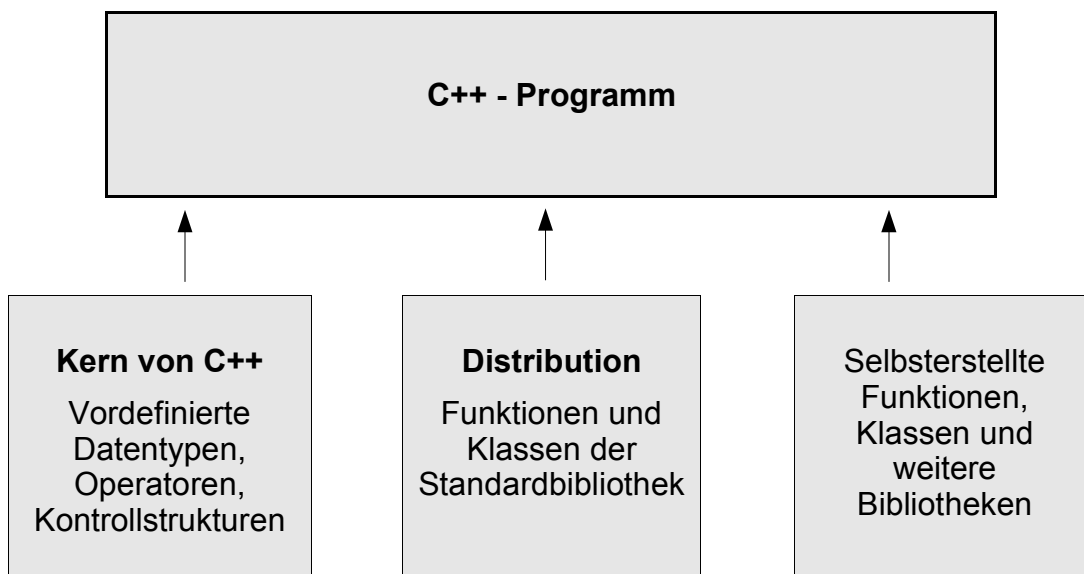


Unterprogramme

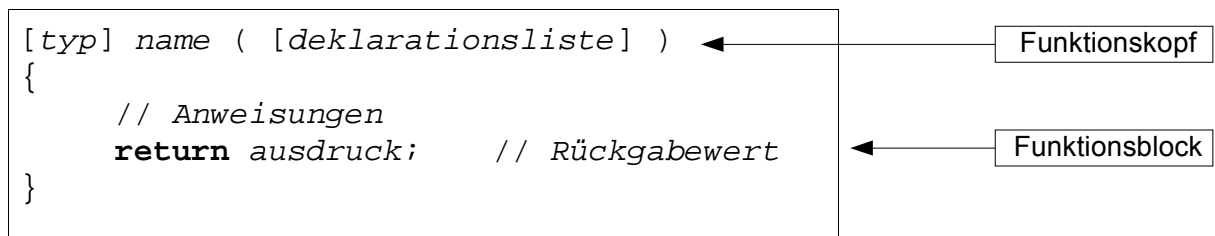
Unterprogramme sind abgekapselte Programmfragmente, welche es erlauben, bestimmte Aufgaben in wiederverwendbarer Art umzusetzen. Man unterscheidet zwischen Unterprogrammen mit Rückgabewert (**Funktionen**) und ohne (**Prozeduren**). In C++ sind Prozeduren nichts weiter als Funktionen mit einem speziellen Rückgabebetyp "void".

Funktionen

Bedeutung von Funktionen in C++



Definition einer Funktion



Definition einer Prozedur

```
void name ( [deklarationsliste] )  
{  
    // Anweisungen  
}
```

Einfaches Beispiel

- Definition der Funktionen:

```
void daten_speichern (char name[20], int gebjahr)
{
    // ... Datenbankspeicherung ...

    // ... Bestätigung ...
    cout << "Folgende Daten gespeichert:" << endl
         << "Name          : " << name          << endl
         << "Geburtsjahr: " << gebjahr        << endl;
}
```

```
int lese_gebjahr (char name[20])
{
    // ... Datenbank abfragen ...
    // ... Ergebnis in Variable "gebjahr" speichern

    return gebjahr;
}
```

- Aufruf in der Programm-Hauptfunktion "main":

```
int main()
{
    char name[20];
    int jahr;

    cout << "Geben Sie ihren Namen ein: ";
    cin >> name;
    cout << "Geben Sie ihr Geburtsjahr ein: ";
    cin >> jahr;

    // Name und Geburtsjahr speichern
    daten_speichern(name, jahr);
    // Geburtsjahr ausgeben
    cout << lese_gebjahr(name) << endl;

    return 0;
}
```

Jede Funktion muß vor ihrer Verwendung deklariert werden, d.h. der *name* und *typ* der Funktion sowie die Typen der Parameter der *deklarationsliste* müssen vorher angegeben werden.

- Probleme:
 - bei vielen selbst erstellten Funktionen wird es schwierig, stets die korrekte Definitions-Reihenfolge beizubehalten
 - wenn sich 2 Funktionen gegenseitig aufrufen wollen, weiß die zuerst definierte nichts von der danach definierten Funktion
 - Lösung:
 - die Anweisung, welche Funktion aufruft, muß nur wissen, wie die Schnittstelle der Funktion aussieht (Rückgabetyt, Parametertypen)
-> Funktionskopf
 - später im Programmcode folgt dann die Definition der Funktion
-> Funktionskopf + Funktionsblock
- => Unterteilung in **Prototypen** (siehe unten) und **Definitionen** (siehe oben)

Prototyp einer Funktion

Der **Prototyp** ist die **Deklaration** einer Funktion, welche die formale Benutzer-Schnittstelle der Funktion beschreibt. Er ist dem **Funktionskopf** der **Funktionsdefinition** ähnlich und unterscheidet sich nur durch nachfolgendem Semikoleon statt **Funktionsblock**.

- Prototypen der o.g. Funktionen:

```
void daten_speichern (char name[20], int gebjahr);  
int lese_gebjahr (char name[20]);
```

- Parameternamen dürfen in Prototypen weggelassen werden:

```
void daten_speichern (char[20], int);  
int lese_gebjahr (char[20]);
```

Erweitertes Beispiel

```
// Prototypen
void daten_speichern (char[20], int);
int lese_gebjahr (char[20]);

// Hauptfunktion
int main()
{
    ...
    return 0;
}

// Definitionen
void daten_speichern (char name[20], int gebjahr)
{
    ...
}

int lese_gebjahr (char name[20])
{
    ...
}
```

Praxisbezug

- in größeren Software-Projekten werden Prototypen in sogenannten “Header-Dateien” und Funktionsdefinitionen in sogenannten “Modulen” gesammelt
- “Header-Dateien” besitzen meist die Endung *.h oder *.hh, “Module” hingegen *.o (object files) oder *.a (object files archive) oder *.so.*versionsnummer* (shared object files)
- “shared object files” werden erst beim Programmstart eingebunden, während alle anderen Module schon zur Compile-Zeit verwendet werden (Plugin-Konzept)
- unter Linux sind Header meist in “/usr/include” und Module meist in “/usr/lib” zu finden
- Vertiefung in später folgenden Praktika

Übergabe von Funktionsparametern

Call By Value

Call By Value bezeichnet die Übergabe von *Werten* beim Aufruf einer Funktion. Die aufgerufene Funktion arbeitet dann mit der Kopie der Argumente, welche lokal gültig ist.

- Beispiel:

```
#include <iostream>
using namespace std;

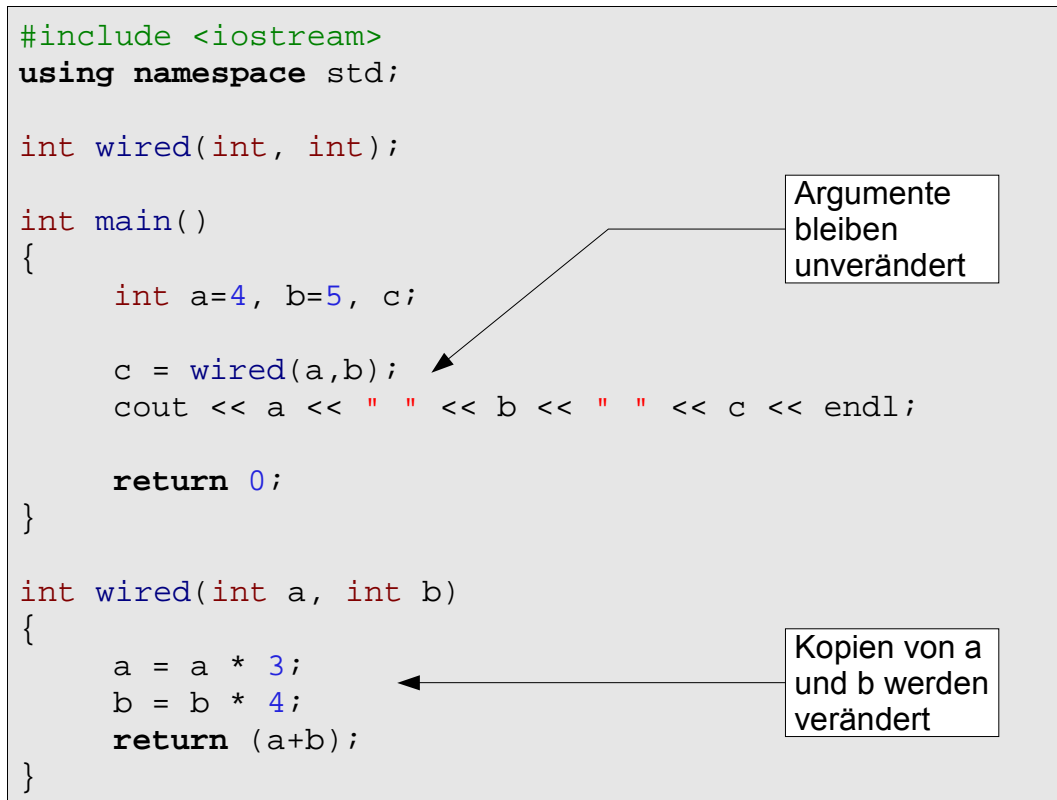
int wired(int, int);

int main()
{
    int a=4, b=5, c;

    c = wired(a,b);
    cout << a << " " << b << " " << c << endl;

    return 0;
}

int wired(int a, int b)
{
    a = a * 3;
    b = b * 4;
    return (a+b);
}
```



- Ausgabe:

4 5 32

Call By Reference

Call By Reference bezeichnet die Übergabe von *Argumentadressen* (Aliasname) beim Aufruf einer Funktion. Die aufgerufene Funktion arbeitet dann direkt mit den Argument-Objekten und kann diese auch verändern.

- in C++ werden Adressen mit dem Ausdruck "&" gekennzeichnet (es gibt auch Möglichkeit, Call By Reference über Zeiger zu realisieren, aber dies folgt im nächsten Semester)

- Beispiel:

```

#include <iostream>
using namespace std;

int wired(int&, int&);

int main()
{
    int a=4, b=5, c;

    c = wired(a,b);
    cout << a << " " << b << " " << c << endl;

    return 0;
}

int wired(int &a, int &b)
{
    a = a * 3;
    b = b * 4;
    return (a+b);
}

```

- Ausgabe:

12 20 32

- sollen Argumente nicht verändert werden:

```
int test (const int &a);
```

Gegenüberstellung der Verfahren

Verfahren	Vorteile	Nachteile
Call By Value	<ul style="list-style-type: none"> • Argumente können beliebige Ausdrücke sein • Funktion kann keine unbeabsichtigten Änderungen der Argumente vornehmen 	<ul style="list-style-type: none"> • durch Kopie-Charakter ist Übergabe großer Objekte rechen- und speicheraufwändig • nur ein Ergebnis über <code>return</code> zurückgebbar
Call By Reference	<ul style="list-style-type: none"> • Argumente werden nicht kopiert, weshalb Übergabe großer Parameter schnell ist • mehrere Werte können über Referenzparameter zurückgegeben werden 	<ul style="list-style-type: none"> • Argumente müssen Objekte im Speicher mit korrektem Typ sein -> beliebige Ausdrücke (z.B. <code>a+b</code>) nicht möglich