Programmieren / Algorithmen & Datenstrukturen

Grundlagen (i), Teil 6

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Grundlagen (i)

Inhalt.

- Hallo C++
- Objekte, Typen, Werte, und Steuerungsprimitive
- Berechnungen und Anweisungen
- Fehler
- Fallstudie: Taschenrechner
- Funktionen und Programmstruktur
- Klassen

Funktionen und Programmstruktur

Einige technische Details zu C++ Funktionen und zur Programmstruktur.

- Wir werden uns noch etwas genauer mit den technischen Einzelheiten von Funktionen in der Programmiersprache C++ befassen.
- Aber: es geht nicht um eine auch nur annähernd vollständige Beschreibung der Syntax und Semantik von C++.
  - Nicht einmal für die Sprachelemente, die wir durchnehmen.
  - Die C++ Sprachbeschreibung umfasst über 1000 Seiten...
- Unser Ziel: hohe Verständlichkeit und hoher Nutzen im Verhältnis zum Aufwand.
  - Wir wollen uns nicht in syntaktischen und semantischen Sprachdetails verlieren.
  - Es gibt auch in C++ meist mehr als eine Möglichkeit, etwas gut zu programmieren (wie man z.B. auch auf Deutsch eine Sache meist auf mehr als eine Weise gut ausdrücken kann).
- Bitte immer daran denken:
  - Unser Interesse liegt auf Programmieren, Algorithmen & Datenstrukturen,
  - die Programmiersprache ist unser Werkzeug.

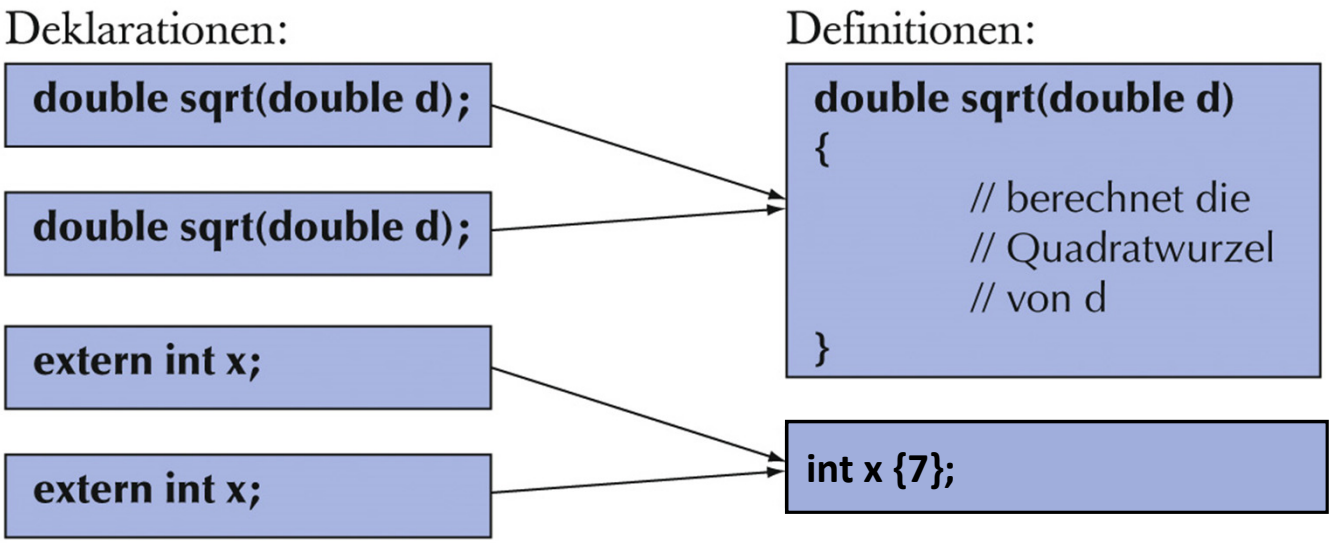
Deklaration und Definition

Diese Unterscheidung ermöglicht es, die *Schnittstelle zum Aufruf* von der *Implementierung* einer Funktion zu trennen.

- Eine *Deklaration* ist eine Anweisung, die einen Namen in einen Gültigkeitsbereich einführt (C++ Grundsatz: Deklaration vor Verwendung) und
  - einen Typ für das benannte Objekt angibt (eine Funktion ist vom Typ ihrer Rückgabe),
  - optional einen Initialisierer (z.B. Literal, Funktionsrumpf) angibt (und dann gleichzeitig eine Definition ist).
  - D.h.: eine Deklaration ist die Schnittstelle zur Verwendung des deklarierten Elements.
    - Meist in Headerdateien zu finden.
    - *Deklarationen können sich (identisch) beliebig oft wiederholen.*
- Eine Deklaration, die das deklarierte Element auch vollständig spezifiziert, nennt man *Definition.*
  - Die Definition reserviert im Unterschied zur Deklaration Speicher.
  - Die Definition ist immer auch eine Deklaration.
  - D.h.: die Definition ist die Implementierung, durch welche das benannte Element das macht, wofür es gedacht ist.
    - *Jeder Name muss genau einmal definiert sein (one definition rule).*

Deklaration und Definition

Schnittstelle und Implementierung.

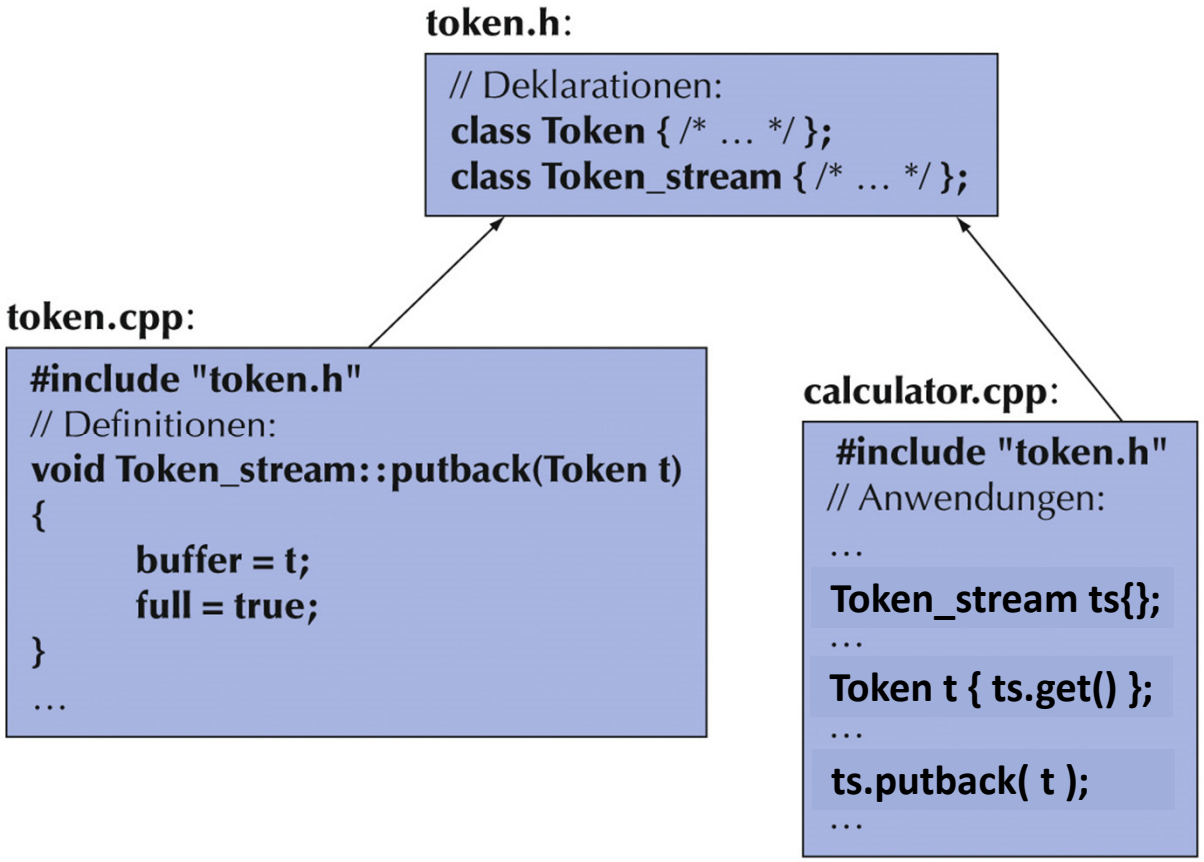


- Durch die Unterscheidung Deklaration-Definition lässt sich das Programm auch in mehrere Teile (Dateien) zerlegen, die voneinander getrennt kompiliert werden.
  - Jeder Teil muss (nur) alle Deklarationen kennen (die konsistent sein müssen).
  - Der Linker verknüpft den Objektcode.
- Das Schlüsselwort `extern`
  - gibt an, dass die nachfolgende Deklaration *keine* Definition ist.

Bildquelle: Stroustrup.

Deklaration und Definition, Headerdateien

Header werden mit der `#include` Direktive vom Präprozessor in eine Quellcode-Datei einkopiert.



Bildquelle: Stroustrup.

Deklaration, Definition und Initialisierung

Alle Variablen sollen grundsätzlich vor ihrer ersten Verwendung initialisiert werden.

- Nicht initialisierte Variablen:

```
int i{}; // initialisiert, {} Syntax (Wertkonstruktion)
double d = 0.0; // initialisiert, = Syntax (Zuweisungsoperator)
vector<int> vi( 10 ); // initialisiert, () Syntax (Konstruktor)

int j; // nicht initialisiert
```

  - Uninitialisiertes `j` nicht typsicher, keine Überprüfung durch den Compiler.
  - Eine häufige Ursache für schwer zu findende Fehler.

- Konstanten:

```
const double pi( 3.1415 ); // initialisiert, () Syntax
//const double e; // Fehler, nicht initialisierte Konstante
const int zwerge{ 7 }; // initialisiert, {} Syntax
```

  - Als `const` deklarierte Namen *müssen* offensichtlich initialisiert werden (da ihnen kein Wert zugewiesen werden kann, Compiler prüft).

Initialisierung durch Standardkonstruktoren

Alle Variablen sollen grundsätzlich vor ihrer ersten Verwendung initialisiert werden.

- Eingebaute Typen (`char`, `bool`, usw.)
  - Werden als lokale Variablen oder Klassenmember *nicht* "von selbst" initialisiert.
  - Verwenden Sie daher zur Initialisierung z.B. die `{}` Syntax: `int i{};`
- Benutzerdefinierte Typen aus der StdLib (`string`, `vector`, usw.)
  - Werden (im Unterschied zu den eingebauten Typen) immer durch spezielle Memberfunktionen, die *Standardkonstruktoren*, initialisiert.
  - Erinnern Sie sich an die bereits behandelten Beispiele.

```
vector<int> v; // ist initialisiert: vector mit null int-Elementen
string s; // ist initialisiert: leerer string
```

- Ihre eigenen benutzerdefinierten Typen (Klassen)
  - Sehen Sie möglichst *immer* Standardkonstruktoren zur Initialisierung vor.
  - Ansonsten sind schwer lokalisierbare Fehler vorprogrammiert.

`auto` Typspezifikation für Variablen

Man muss den Typ einer Variablen nicht ausdrücklich angeben, er kann auch aus der Initialisierung abgeleitet werden.

- Beispiele

```
auto b = true;    // b ist vom Typ bool
auto i = 42;      // i ist vom Typ int
```

  - Die beiden Beispiele verdeutlichen zwar das Prinzip, sind aber ansonsten kein wirklicher Einsatzbereich für `auto`.
  - Nützlich wird diese Syntax erst, wenn ein Typname schwierig zu schreiben bzw. herauszufinden ist, ein Beispiel folgt gleich beim Thema `bind()`.

- Zur Vermeidung von Komplikationen und Überraschungen verwendet man mit `auto` *nur* die Initialisierung mittels **=** Syntax, d.h. durch Zuweisungsoperator wie oben.

```
auto b2 {true};   // b2 ist vom Typ initializer_list<bool>
auto i2 {42};     // i2 ist vom Typ initializer_list<int>
```

- Weiteres Beispiel

```
double g( int& v ) {
  auto x = v;   // x ist vom Typ int (nicht int&)
  auto& y = v;  // y ist vom Typ int&
}

auto z = g( i ); // z ist vom Typ double;
```

Gültigkeitsbereich ("Scope")

Gültigkeitsbereiche kann man sich als bestimmte Abschnitte im Quellcode vorstellen.

- Namen werden innerhalb eines Gültigkeitsbereichs deklariert und sind dann vom Punkt ihrer Deklaration bis zum Ende des Gültigkeitsbereichs "im Scope".
- *Globaler Bereich*
  - Liegt außerhalb jedes anderen Scopes.
- *Namensraum*
  - Explizit benannter Gültigkeitsbereich (`namespace`), liegt im globalen Scope oder innerhalb eines anderen Scopes.
- *Klassenbereich*
  - Quellcode innerhalb einer Klasse.
- *Lokaler Bereich*
  - Quellcode zwischen den geschweiften Klammern `{ }` eines Blocks (bzw. ab der Argumentliste einer Funktion).
- *Anweisungsbereich*
  - Quellcode innerhalb einer (einzigen) Anweisung, z.B. `for`-Anweisung.

Gültigkeitsbereich ("Scope")

Gültigkeitsbereiche kann man sich als bestimmte Abschnitte im Quellcode vorstellen.

- Die Hauptaufgabe des Gültigkeitsbereichs ist es, Namen lokal zu halten, damit sie nicht mit anderen Namen kollidieren.
  - Mit Ausnahme des globalen Gültigkeitsbereichs hält ein Gültigkeitsbereich also Namen lokal.
- Sinn:
  - Reale Programme können schnell Hunderttausende von benannten Elementen enthalten.
  - Gibt es viele globale Namen, ist es praktisch nicht mehr möglich, in solchen Programmen noch zu wissen, welche Elemente mit einem bestimmten globalen Namen zu tun haben.
    - Das ist auch der Hauptgrund dafür, dass Sie in Ihren Programmen schon jetzt keine globalen Variablen einsetzen sollen.
  - Namen müssen also lokal bleiben, damit man nicht den Überblick verliert.

Gültigkeitsbereich ("Scope")

Gültigkeitsbereiche sind meist verschachtelt.

- Beachten Sie, dass die meisten C++ Konstrukte, die Gültigkeitsbereiche festlegen, selbst wieder in andere Gültigkeitsbereiche eingeschlossen (verschachtelt) sind.
- *Memberfunktionen* (d.h. Funktionen in Klassen, auch *Methoden* genannt)
  - Häufiger und üblicher Fall.
- Memberklassen (d.h. Klassen in Klassen)
  - Typisch für die Implementierung fortgeschrittener Klassen.
- Lokale Klassen (d.h. Klassen in Funktionen)
  - Vermeiden Sie diese Konstruktion.
  - Wenn Sie das Gefühl haben, eine lokale Klasse zu brauchen, ist vermutlich Ihre Funktion (viel) zu umfangreich.
- Lokale Funktionen (d.h. Funktionen in Funktionen)
  - In C++ nicht erlaubt (außer sog. Lambdas, die streng genommen keine Funktionen sind).

Gültigkeitsbereich ("Scope")

Gültigkeitsbereiche sind meist verschachtelt.

- *Blöcke* (Quellcodeabschnitte zwischen geschweiften Klammern)
  - Als Funktionen
    - Üblicher Fall.
  - In anderen Blöcken ("verschachtelt")
    - Nicht zu vermeiden.
    - Versuchen Sie, verschachtelte Blöcke in Ihren Programmen nicht zu kompliziert werden zu lassen, da solche Konstruktionen anfällig für Fehler sind.
- Scope-Auflösung durch die `::` Syntax möglich.

```
int errno{}; // globale Variable
void f( ) { int i{::errno}; } // Direktzugriff auf
                              // globales errno.
```

Gültigkeitsbereich ("Scope")

Benannte Namensbereiche (`namespace`).

- Blöcke organisieren den Quellcode innerhalb von Funktionen.
- Klassen organisieren Funktionen und Daten als Typen.
- *Namensräume* organisieren Klassen, Funktionen, Daten und Typen zu identifizierbaren, benannten Teilen des Programms (ohne einen extra Typ definieren zu müssen).

```
namespace TextLib {
  class Text { /* … */ }
  class Line { /* … */ }
  class Hyphenation { /* … */ }
  // …
}
namespace GraphLib {
  class Point { /* … */ }
  class Line { /* … */ }
  class Shape { /* … */ }
  // …
}
```

```
// using-Deklaration
using GraphLib::Line;
obj1 = Line( 1 );


// vollqualifizierter Name
obj2 = TextLib::Line( 60 );

// KEINE Namenskonflikte
// mit Line
```

Gültigkeitsbereich ("Scope")

Benannte Namensbereiche (`namespace`).

- Sehen Sie sich den Quellcode von zwei Programmierern an:

Julia.h

```
class Jobj { /*…*/ };      // Julias header
class Widget { /*…*/ };    // Julias header
```

Romeo.h

```
class Robj { /*…*/ };      // Romeos header
class Widget { /*…*/ };    // Romeos header
```

fbi.cpp

```
#include "Julia.h";    // wird einkopiert
#include "Romeo.h";    // wird einkopiert

// Fehler: Name Widget mehrdeutig

void my_func( Widget p )
{
    // …
}
```

Gültigkeitsbereich ("Scope")

Benannte Namensbereiche (`namespace`).

- Konfliktvermeidung mit `namespace` Definitionen.

Julia.h

```
namespace Julia {
    class Jobj { /*…*/ };
    class Widget { /*…*/ };
}
```

Romeo.h

```
namespace Romeo {
    class Robj { /*…*/ };
    class Widget { /*…*/ };
}
```

fbi.cpp

```
#include "Julia.h";    // wird einkopiert
#include "Romeo.h";    // wird einkopiert

void my_func( Julia::Widget p )   // OK
{
    // …
}
```

Funktionsaufrufe und -rückgabewerte

Deklaration einer Funktion durch Angabe von Name, Rückgabetyp sowie Reihenfolge und Typen der Parameter.

- Eine Funktionsdeklaration beginnt mit dem *Rückgabetyp*, gefolgt vom *Name* der Funktion und der in Klammern gefassten *Liste der Parametertypen*.

```
double fct( int, double );       // Deklaration (kein Rumpf)
double fct( int i, double d ) { // Definition
  return i*d;
}

void setPower( int level );  // kein Rueckgabewert
int getPower( );             // keine Parameter
```

- Parameter*namen* in Deklarationen werden von C++ ignoriert.
  - Sie können weggelassen werden.
  - Sie sollten aber (zur besseren Lesbarkeit für Menschen) angegeben werden.

Funktionsaufrufe und -rückgabewerte

Deklaration einer Funktion durch Angabe von Name, Rückgabetyp sowie Reihenfolge und Typen der Parameter.

- Um einen Wert aus einer Funktion zurück zu geben wird die `return` Anweisung verwendet.

```
double abs( int x ) {              // liefert double
  if( x < 0 ) return -x;
  if( x > 0 ) return x;
}
// Fehlerhafter Quellcode: was ist mit x==0 ?
```

- Das Zurückliefern eines Werts durch eine Funktion ist eine Form der Initialisierung.
- Stellen Sie daher sicher, dass jeder denkbare Weg aus der Funktion heraus zu einer `return` Anweisung führt oder eine Ausnahme auslöst.
- Sonderfall `main()`.
  - Ist das Ende von `main()` erreicht, ist dies gleichbedeutend mit `return 0;`
- Sonderfall `void` Rückgabe (d.h. die Funktion liefert keinen Wert zurück).
  - Aus einer solchen Funktion wird mit `return;` zurückgekehrt.
  - Ist das Ende einer solchen Funktion erreicht, ist dies gleichbedeutend mit `return;`

Funktionsaufrufe und -rückgabewerte

"Pass-by-value" Parameterübergabe bzw. Ergebnisrückgabe durch Kopieren des Werts.

- Die Parameter beim Funktionsaufruf werden als *Kopien* der Objektwerte übergeben.
  - D.h. sie sind in der aufgerufenen Funktion lokal und werden bei jedem Aufruf neu mit dem jeweils übergebenen Wert initialisiert (entspricht dem Prinzip, dass eine Funktion ihre Aufrufparameter nicht ändern soll).
  - Auch Rückgabewerte werden normalerweise als Kopien durch Umspeichern zurück gegeben.

```
int f( int a ) { a = a+1; return a; }

int main( ) {
   int xx{};
   cout << f(xx) << endl;     // gibt 1 aus
   cout << xx << endl;        // gibt 0 aus, f() aendert xx nicht
   int yy{7};
   cout << f(yy) << endl;     // gibt 8 aus
   cout << yy << endl;        // gibt 7 aus, f() aendert yy nicht
   //…
}
```

- Sehen Sie sich dazu auch nochmals die Grundlagen (i), Teil 3 an.

Funktionsaufrufe und -rückgabewerte

"Pass-by-reference" Parameterübergabe bzw. Ergebnisrückgabe durch einen Alias für das übergebene Objekt.

- L-Referenzen als Parameter beim Funktionsaufruf sind *weitere Namen* für die übergeben Objekte (d.h. sie verweisen auf die Objekte).
  - Eine Funktion kann mit dieser Technik direkt auf einem beliebigen Objekt operieren, was zu schwer lokalisierbaren Fehlern führen kann.
  - Verwenden Sie diese Art des Funktionsaufrufs also nur, wenn es dafür gute Gründe gibt (was durchaus vorkommt).

```
int f( int& a ) { a = a+1; return a; }

int main( ) {
   int xx{};
   cout << f(xx) << endl;     // gibt 1 aus, f() aendert xx
   cout << xx << endl;        // gibt 1 aus
   int yy{7};
   cout << f(yy) << endl;     // gibt 8 aus, f() aendert yy
   cout << yy << endl;        // gibt 8 aus
   //…
}
```

- Sehen Sie sich dazu auch nochmals die Grundlagen (i), Teil 3 an.

Funktionsaufrufe und -rückgabewerte

"Pass-by-`const`-reference" Parameterübergabe bzw. Ergebnisrückgabe durch einen Alias, durch den das Objekt nicht geändert werden kann.

- Sinn: Übergabe von großen Objekten (lange Zeichenketten, große Bitmaps, usw.) an Funktionen, ohne sie zeit- und speicheraufwändig zu kopieren.
- Wert, L-Referenz und `const`-L-Referenz im Beispiel.

```
void f( int a, int& r, const int& cr ) {
   ++a; ++r; ++cr; // geht nicht: cr ist const
}
void g( int a, int& r, const int& cr ) {
   ++a; ++r; int x = cr; ++x;
}
int main( ) {
   int x{}; int y{}; int z{};
   g( x, y, z );  // x==0 y==1 z==0
   g( 1, 2, z );  // geht nicht, r muss Variable sein
   g( 1, y, 3 );  // cr ist const
   //…
}
```

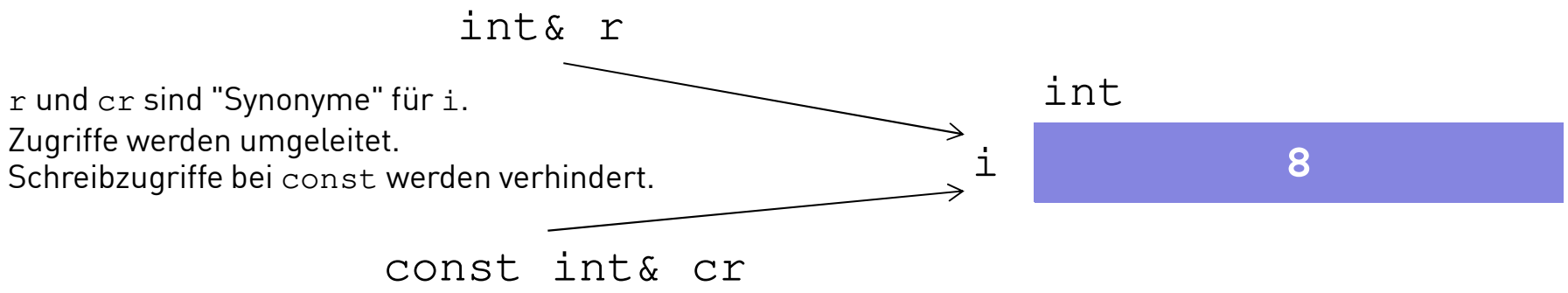
- Sehen Sie sich dazu auch nochmals die Grundlagen (i), Teil 3 an.

L-Referenzen

L-Referenzen sind ein allgemeines Sprachmittel in C++, das nicht nur für Parameter in Funktionsaufrufen einsetzbar ist.

- Man kann sich eine L-Referenz als Synonym oder Alias, als neu deklarierten Namen für das bestehende Objekt vorstellen.

```
int i{7};
int& r{i};
r = 9;  // i wird 9
const int& cr{i};
cr = 7; // geht nicht: cr ist const
i = 8;
cout << cr << endl;   // gibt den Wert von i aus (also 8)
```



L-Referenzen

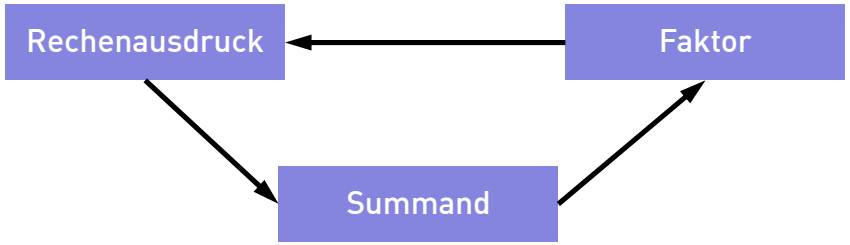
L-Referenzen sind ein allgemeines Sprachmittel in C++, das nicht nur für Parameter in Funktionsaufrufen einsetzbar ist.

- L-Referenzen *müssen* initialisiert werden.
  - Mit einem L-Wert, also mit etwas, das eine Speicheradresse besitzt.
    - Im einfachsten Fall mit einer Variablen.
    - Ist die rechte Seite der L-Referenzdeklaration ein Ausdruck ohne Speicherplatz, dann ist die Referenzbildung sinnlos, manche Compiler lassen es aber zu.
  - Mit der Initialisierung wird ein zusätzlicher Name an einen existierenden Speicherplatz gebunden.
    - D.h. eine L-Referenz ist tatsächlich nie selbst ein Operand (obwohl es im Quellcode anders aussieht).
  - Die Zuordnung einer Referenz lässt sich nicht mehr ändern.
- *Ausblick:* L-Referenzen ähneln dem Verfahren der Adressübergabe (Zeigertypen), haben aber den Vorteil, dass das ständige De-Referenzieren der Zeiger mit dem Operator * entfällt.

Implementierung von Funktionsaufrufen

Aktivierungsdatensätze (function activation records) am Beispiel von `rechenausdruck()`, `summand()` und `faktor()`.

- Sie erinnern sich an die drei Funktionen aus den Grundlagen, Teil 5 (Fallstudie Mini-Rechner).

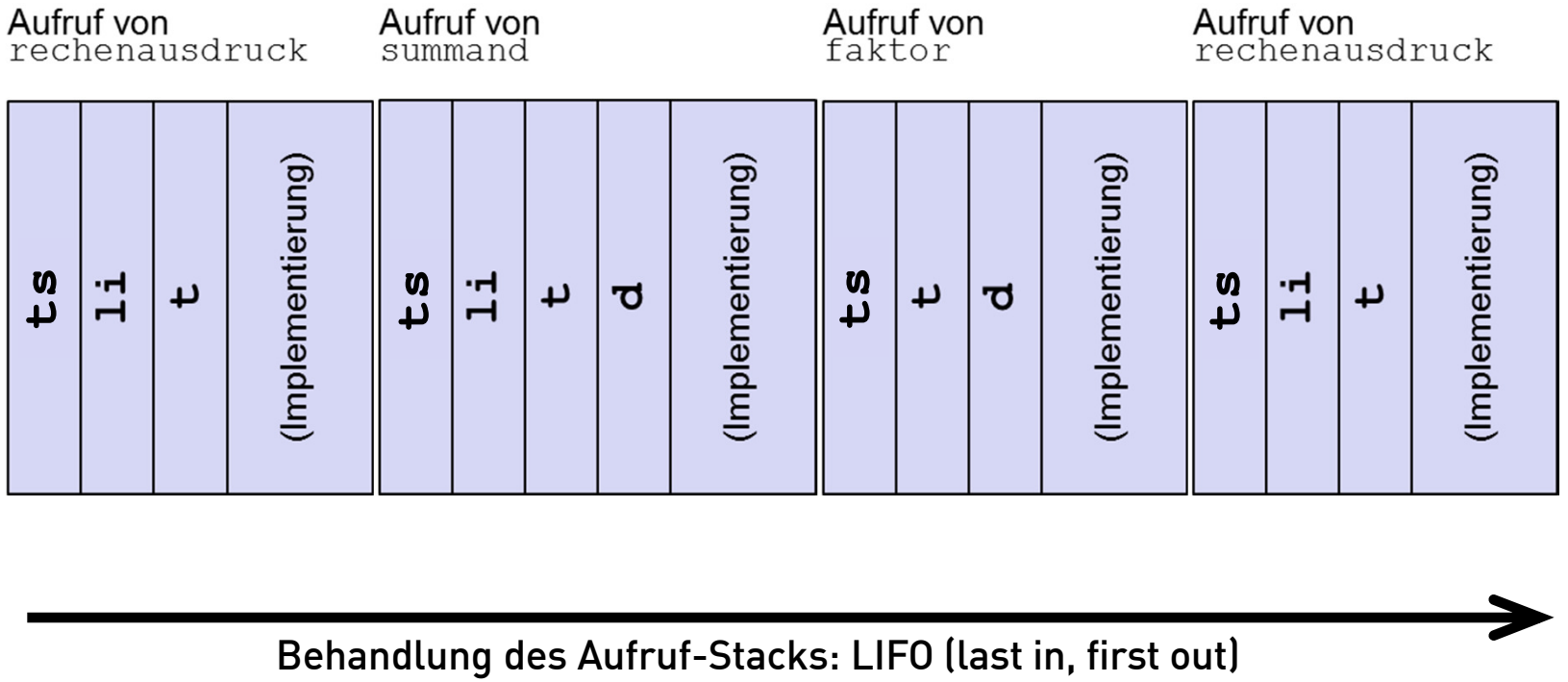



```
double rechenausdruck( Token_stream& ts );
double summand( Token_stream& ts );
double faktor(  Token_stream& ts );
```

- Wird eine Funktion in C++ aufgerufen, richtet die Sprachimplementierung für den Aufruf eine Datenstruktur ein, in der alle zugehörigen Parameter und lokalen Variablen enthalten sind.
- Diese Datenstruktur heißt *Function Activation Record (FAR)*.
- Mit Ende der Funktion verschwindet der komplette FAR wieder.

Implementierung von Funktionsaufrufen

Die Aktivierungsdatensätze werden im sog. Stack (ein bestimmter Speicherbereich) nach dem LIFO-Prinzip abgespeichert.



Auswertungsreihenfolge

Bei der Programmausführung werden Anweisungen nach den Regeln der Sprache der Reihe nach abgearbeitet.

- Anlegen und Auflösen von Variablen in ihrem Scope.
  - Lokale Variablen werden jedesmal beim Eintritt des Programmablauf-Pfads in einen Gültigkeitsbereich angelegt (in der definierten Reihenfolge) und beim Austritt aus dem Gültigkeitsbereich aufgelöst (in umgekehrter Reihenfolge).
- Prioritäten von Operatoren sind zu beachten.
  - Vermeiden Sie Konstruktionen wie in den folgenden Beispielen, da jeweils die Auswertungsreihenfolge undefiniert ist.

```
v[i] = ++i;
v[++i] = i;
int x { ++i + ++i };
cout << ++i << ' ' << i << endl;
f( ++i, ++i );
```

Vermeiden Sie solche undefinierten Ausdrücke.

- Grundregel: wenn Sie den Wert einer Variablen in einem Ausdruck verändern, greifen Sie in diesem Ausdruck nicht ein weiteres Mal auf die Variable zu.

Der `bind()` Funktionsadapter

`std::bind()` (Header `functional`) erwartet als ersten Parameter eine Funktion, die dann mit den übrigen Parametern aufgerufen wird.

- Einfaches Beispiel

```
double sqrt( double ); // gibt Wurzel des Parameters zurueck
auto s = bind( sqrt, 2 ); // 1.4142
```

  - Jeder Aufruf `s()` wird `sqrt(2)` ausführen.
  - `bind()` gibt einen sog. Funktortyp zurück, mehr dazu folgt noch.

- Wirklich nützlich wird es erst, wenn nicht alle Parameter gebunden sind.
  - Dafür kann man den `placeholders` Mechanismus der StdLib (auch im Header `functional`) einsetzen, Beispiel:

```
double pow( double base, double exponent );

using std::placeholders::_1;

auto pow2 = bind( pow, _1, 2.0 ); // _1 hoch 2
cout << pow2(3); // 9

auto pow10 = bind( pow, 10.0, _1 ); // 10 hoch _1
cout << pow10(3); // 1000
```

Einige Beispielfragen

Funktionen und Programmstruktur.

- Was ist der Unterschied zwischen einer Deklaration und einer Definition?
- Wie unterscheidet sich die Syntax einer Funktions- / Variablendeklaration von der Syntax einer Funktions- / Variablendefinition?
- Ist `int i;` eine Definition oder eine Deklaration? Warum? Falls Sie es für eine Definition halten, wie lautet die entsprechende Deklaration? Falls Sie es für eine Deklaration halten, wie lautet die entsprechende Definition?
- Warum ist es dringend ratsam, alle Variablen direkt mit ihrer Definition zu initialisieren?
- Welche Funktion haben Headerdateien?
- Was ist der Gültigkeitsbereich einer Deklaration?
- Welche Gültigkeitsbereiche kennen Sie? Geben Sie jeweils ein *eigenes* Beispiel.
- Wozu gibt es benannte Namensbereiche? Wie lautet das entsprechende C++ Schlüsselwort für die Definition eines solchen Namensbereichs?

Einige Beispielfragen

Funktionen und Programmstruktur.

- Geben Sie ein Beispiel für eine undefinierte Auswertungsreihenfolge. Warum kann so etwas zu einem Problem werden?
- Diskutieren Sie "pass-by-value", "pass-by-reference" und "pass-by-`const`-reference" am einem *eigenen* Beispiel.
- Ist es möglich, ein Funktionsargument (das keine L-Referenz ist) als `const` zu deklarieren?
  - Etwa `void func( const int );`
  - Wann sollte man das machen?
  - Warum wird es nicht oft gebraucht?

  Versuchen Sie es, schreiben Sie einige Quellcodezeilen und probieren Sie aus, was passiert.
- Die folgenden Quellcodes beschreiben in einem einleitenden Kommentar eine Aufgabenstellung und lösen diese dann:
  - Sehen Sie sich die Quellcodes und die Kommentare genau an.
  - Probieren Sie die Quellcodes aus.
  - Was kann man aus diesen Quellcodes über Funktionen lernen?

Quellcode-Beispiele

Funktionen und Programmstruktur.

```
/* Write a function that takes a vector<string> argument and returns
   a vector<int> containing the number of characters in each string.

   Also find the longest and shortest string and the lexicographically
   first and last string.

   How many separate functions would you use for these tasks? Why? */
```

```
vector<int> get_sizes( const vector<string>& vs )
{
   vector<int> res( vs.size() );
   for( int i{0}; i < vs.size(); ++i )
       res[i] = vs[i].size();
   return res;
}
```

Quelle: Stroustrup.

Quellcode-Beispiele

Funktionen und Programmstruktur.

```
int longest( const vector<string>& v )
{
   if( v.size()==0 ) error( "longest(): empty vector" );   // protect against
                                                           // the empty string

   int m{ v[0].size() };     // we now know that there is a v[0]
   int max_index{ 0 };
   for( int i{0}; i < v.size(); ++i )
       if( m < v[i].size() ) {
           max_index = i;
           m = v[i].size();
       }
   return max_index;
}
```

Quelle: Stroustrup.

Quellcode-Beispiele

Funktionen und Programmstruktur.

```
string& shortest( vector<string>& v )
{
   if( v.size()==0 ) error( "shortest(): empty vector" );  // protect against
                                                           // the empty string

   int m{ v[0].size() };     // we now know that there is a v[0]
   int min_index{ 0 };
   for( int i{0}; i < v.size(); ++i )
       if( v[i].size() < m ) {
           min_index = i;
           m = v[i].size();
       }
   return v[min_index];
}
```

Quelle: Stroustrup.

Quellcode-Beispiele

Funktionen und Programmstruktur.

```
string lex_first( const vector<string>& v )
{
   if( v.size()==0 ) return "";  // the empty string is first

   int first_index{ 0 };
   for( int i{0}; i < v.size(); ++i )
       if( v[i] < v[first_index] )
           first_index = i;
   return v[first_index];
}
```

Quelle: Stroustrup.

Quellcode-Beispiele

Funktionen und Programmstruktur.

```
void lex_last( vector<string>& v, int& last )
{
   if( v.size()==0 ) {
        last = -1;   // -1 indicates "the empty vector"
        return;
   }

   int last_index{ 0 };
   for( int i{0}; i < v.size(); ++i )
        if( v[last_index] < v[i] )
            last_index = i;
   last = last_index;    // copy last_index out of function
}
```

Quelle: Stroustrup.

Nächste Einheit:
Klassen